

Europäisches Patentamt  
European Patent Office  
Office européen des brevets



(11) **EP 1 014 265 A1**

(12) **EUROPEAN PATENT APPLICATION**

(43) Date of publication:  
**28.06.2000 Bulletin 2000/26**

(51) Int Cl.7: **G06F 11/00**

(21) Application number: **99309818.5**

(22) Date of filing: **07.12.1999**

(84) Designated Contracting States:  
**AT BE CH CY DE DK ES FI FR GB GR IE IT LI LU  
MC NL PT SE**  
Designated Extension States:  
**AL LT LV MK RO SI**

- **Lane Thompson, Kenneth**  
**Watchung, New Jersey 07060 (US)**
- **Winterbottom, Philip Steven**  
**Watchung, New Jersey 07060 (US)**

(30) Priority: **15.12.1998 US 211967**

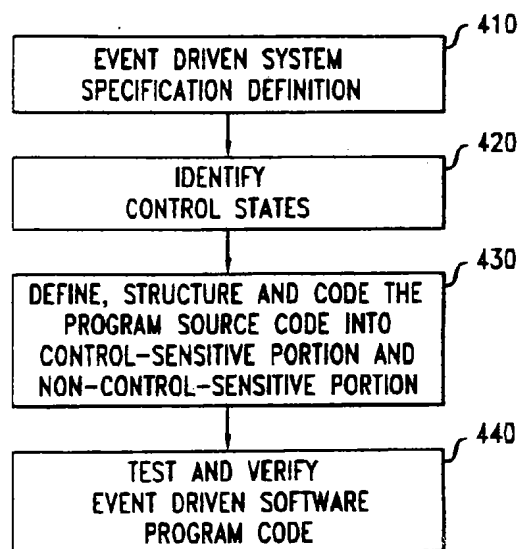
(74) Representative:  
**Watts, Christopher Malcolm Kelway, Dr. et al**  
**Lucent Technologies (UK) Ltd,**  
**5 Mornington Road**  
**Woodford Green Essex, IG8 0TU (GB)**

(72) Inventors:  
• **Holzmann, Gerard Joan**  
**Murray Hill, New Jersey 07974 (US)**

(54) **Method and apparatus for testing event driven software**

(57) A technique for testing event driven software. In accordance with the technique, the source code of the event driven software is directly converted to a automaton based model useful in verifying that the program code complies with the desired properties defined by the user. More particularly, the event driven system program code is translated into a target language for a particular model checker. Such a translation results in a model which contains statements directed at whether execution of the program code will affect the behavior of the event driven system. Thus, this model extraction process can be used as input to a logic model checker for determining whether event driven system complies with the desired correctness properties specified by the user. Advantageously, the model extraction process and application of the model checker occurs in a direct and dynamic fashion from the subject event driven system program code without the need for user intervention.

**FIG. 4**



**Description****Field of the Invention**

5 [0001] The present invention relates to testing computer software programs and, more particularly, to a technique for testing event driven application programs.

**Background of the Invention**

10 [0002] In well-known event driven software systems (also referred to in the art as "reactive systems"), certain functionality occurs when an event is generated and communicated to an executing process performing particular actions based upon the occurrence of the particular event. More particularly, event driven software is used to define the behavior of a system, e.g., a computer, in response to events that are generated by other systems, machines or peripheral devices connected thereto. Well-known examples of event driven systems include: telecommunications applications, 15 call-processing software, data communications software, device drivers, and computer interface applications. As will be appreciated, these types of event driven systems are state oriented. That is, in each state of a system, one specific event from a predefined set of events is expected to occur, and the system is designed to respond to the event in a well-defined manner.

20 [0003] Briefly, as is well-known, a state machine converts a time series of event stimuli or input data through some function into a time series of responses or output data. For example, in an event driven system, at each point in time, each machine is in a particular so-called "control state" which defines the events that are expected to be generated by other machines, and the type of responses that are to be generated upon the arrival of each such event. Typically, the response includes the execution of a finite piece of program code, the generation of new events, and the identification of a new control state for the particular machine responding to the incoming events. Thus, the state machine 25 model of the event driven system plays a critical role in the development of the application program code by the programmer for the system.

30 [0004] Typically event driven application software is written in higher level general-purpose programming languages such as the well-known "C" or "C++" programming languages, or a special-purpose language such as the well-known ITU Systems Description Language ("SDL"). In developing the actual program code for such systems it is common to describe the behavior of the relevant machines in terms of state machines. Using a higher level programming language, e.g., C, software programmers will write event driven software code for the particular application. Typically, the software 35 program development begins by reviewing a so-called system specification which provides a logical description of the desired operations for the application. Working from the specification, one or more programmers will write the necessary code to implement the operations.

40 [0005] This conventional programming process results in an application program which contains no particular visual context amongst and between the various control states of the event driven system. That is, the program code can be viewed as one large "flat" file with a series of program instructions. Further, as the original specification changes during program development, e.g., additional functionality is added to the system, the requisite program code and additional control states are typically added to the source program in a somewhat arbitrary order. In order for a programmer to 45 understand the logical flow from control state-to-control state directly from the program source code it is necessary to manually trace the execution thread through the source, e.g., tracing branch instructions or "goto" statements. As will be appreciated, the tracing of such logical flow, particularly in very large source code programs, is a tedious and time consuming task. This also leads to certain complexity in the testing of the software for so-called "bugs" and to determine whether the application program implements the desired features.

50 [0006] More particularly, a critical state in the development of the application software code for such systems is the testing of the code to validate the properties of the software and to insure that the behavior of the system complies with the design objectives. There are a number of well-known methods for testing event driven software. For example, Boris Beizer, *Black-Box Testing: Techniques for Functional Testing of Software and Systems*, John Wiley and Sons, 1984, describes a well-known testing arrangement wherein a set of test goals is defined from which a limited set of 55 test sequences is deduced. Typically, the set of test sequences is defined through a manual process by the testing engineer through examination of the system design. The set of test sequences is applied to the event driven software under test and either a "pass" or "fail" is recorded as the testing result. This testing methodology is typically applied to general software testing and is well-known in the art to be prone to human error and incomplete. In particular, the number of possible executions of the event driven software under test is incredibly large. In contrast, the number of tests that can be performed on the software with real-time executions is limited by certain practical constraints such as the minimum time required to perform a single test from the total number of tests. As such, it is typical that no more than a few hundred tests can be selected and applied in accordance with this methodology. This results in a very low confidence level in the testing coverage.

[0007] A second event driven software method known in the art is described, e.g., by D. Lee et al, Principles and Methods of Testing Finite State Machines - A Survey, *Proceedings of the IEEE*, Vol. 84, pp. 1090-1126, 1996. In accordance with this testing method, an abstract state machine model for the event drive software program is constructed. The state machine model typically captures only a high-level description of the program code thereby leaving out most of the program details. Thereafter, well-known test sequence generation algorithms (see, e.g., Lee, *supra.*) can be applied to the state machine model from which a relatively small set of tests is derived and applied to the software under test. As with the above-described testing method, this technique relies heavily on the manual construction of the abstract model of the event driven software. Therefore, this technique is sensitive to certain errors caused by mismatches between the formal relation between the software under test and the abstract model to be used for deriving the applicable tests. In addition, changes to the event driven software require the development of new tests which proves cumbersome and time consuming.

[0008] In a third commonly used testing method, see, e.g., C. H. West, Protocol Validation by Random State Exploration, *Proc. 6<sup>th</sup> IFIPWG 6.1 Int. Workshop on Protocol Specification, Testing, and Verification*, North-Holland Publ., pp. 233-242, 1986, a random exploration of the full behavior of the event driven software is undertaken in order to establish the software's compliance with the desired properties. The amount of detail in the full software description generally precludes any attempt to perform an exhaustive exploration of the software for testing purposes. Therefore, well-known statistical algorithms are used to extract and analyze random samples of the software's behavior. However, as with the other above-described testing arrangements, one recognized drawback in this testing approach is that the amount of extraneous detail in the full software description prevents an efficient exhaustive exploration of the full behavior of the software for compliance with high-level properties.

[0009] Therefore, a need exists for a technique that mitigates the above-described problems in the art and improves the programming development and testing of event driven software.

### **Summary of the Invention**

[0010] The present invention provides a technique for testing event driven software directed to the efficient testing of event driven software and making it possible to track the validity of future changes to the program code, as part of the normal maintenance, extension and revision process to the source code. More particularly, in accordance with the preferred embodiment of the invention, event driven software code is specified and written in accordance with the principles of the co-pending, commonly assigned patent application, entitled "Method and Apparatus For Developing Event Driven Software", Application Serial No. , filed on event date herewith. That is, in accordance with the preferred embodiment, the source code program is defined and structured into control-sensitive and non-control-sensitive format. In accordance with the present invention, the event driven source code program is parsed to extract the control states defined in the source program, and to convert the source code into an intermediate state machine format. Thereafter, in accordance with the invention, the intermediate state machine format is converted into a automata-based format for modeling checking purposes.

[0011] In particular, in accordance with the preferred embodiment, the intermediate state machine format is converted into an annotated automaton model. Illustratively, the annotated automaton model is a verification program written in the input language of a specific model checker. Importantly, in accordance with the invention, each statement from the source code program is mapped onto an abstraction within the automaton model. The mapping is achieved using a translation map which is defined at the initial stage of the verification process. The map need not be revised thereafter unless new types of instructions are introduced, e.g., after revisions are made to repair programming faults, into the source program. The mapping, *inter alia*, dictates a fixed translation of the particular statement into the target language of the model checker. Thus, instructions that appear in the map are systematically converted into the automaton model wherever such instructions appear in the source code.

[0012] In addition to the above-described map, in accordance with the invention, a so-called environment model is defined which encapsulates a minimal set of assumptions that must be made about the particular operating environment in which the event driven application is executed. More particularly, as the model checker input language is provided to the model checker, the environmental model is applied. As a result, in accordance with the preferred embodiment of the invention, the verification of the properties of the event driven system is made subject to the environmental model during the testing and checking of the event driven software by the verification system. Thereafter, the testing and checking results are output, in a well-known manner, by the model checker for determining whether the event driven system conforms with the user's desired execution properties and behaviors.

[0013] Advantageously, in accordance with the invention, the model extraction process and application of the model checker occurs in a direct and dynamic fashion from the event driven system program code without the need for user intervention.

**Brief Description of the Drawings****[0014]**

FIG. 1 is a block diagram of an illustrative event driven system;  
 FIG.'s 2 and 3 show an example of a conventional C source code program for an illustrative telecommunications call processing application useful in the illustrative event driven system of FIG. 1;  
 FIG. 4 shows a flowchart of illustrative operations for software development in accordance with the principles of the invention;  
 FIG.'s 5 and 6 show an illustrative C source code program, structured and defined in accordance with the software development operations of FIG. 4, for the illustrative telecommunications call processing application of FIG.'s 2 and 3;  
 FIG.'s 7 and 8 show a translated source code program derived, in accordance with the principles of the invention, from the illustrative C source code program of FIG.'s 5 and 6;  
 FIG. 9 shows a flowchart of illustrative operations for testing event driven software in accordance with the present invention;  
 FIG. 10 shows intermediate state machine code derived from the illustrative C source code program of FIG.'s 5 and 6;  
 FIG. 11 shows an illustrative map for the event driven system defined by the illustrative C source code program of FIG.'s 5 and 6;  
 FIG.'s 12 and 13 show model checker input language derived from applying the illustrative map of FIG. 11;  
 FIG. 14. shows an illustrative set of library functions for use with the model checker input language of FIG.'s 12 and 13;  
 FIG. 15 shows an illustrative environmental model for the telecommunications processing application described in FIG.'s 5 and 6;  
 FIG. 16 shows an illustrative automaton model for a particular control state defined by the telecommunications processing application of FIG.'s 5 and 6;  
 FIG. 17 shows an illustrative full automaton model of the complete telecommunications processing application described in FIG.'s 5 and 6;  
 FIG. 18 shows an illustrative computer system 1800 useful in executing event driven software code which is specified and written in accordance with the present invention; and  
 FIG. 19 shows an illustrative error trace, generated in accordance with the principles of the invention, for the telecommunications processing application described in FIG.'s 5 and 6.  
 Throughout this disclosure, unless otherwise noted, like elements, blocks, components or sections in the figures are denoted by the same reference designations.

**Detailed Description**

**[0015]** A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all other rights with respect to the copyrighted works whatsoever.

**[0016]** The present invention provides a technique for testing event driven software. In accordance with various aspects of the invention, the event driven system program source code is directly converted to a automaton based model useful in verifying that the program code complies with the desired properties defined by the user. More particularly, in accordance with a preferred embodiment of the invention, the event driven system program code is translated into a target language for a particular model checker. Such a translation results in a model which contains statements directed at whether execution of the program code will affect the behavior of the event driven system, e.g., the execution of the program code causes new events to be generated. Thus, this model extraction process of the preferred embodiment of the invention can be used as input to a logic model checker for determining whether event driven system complies with the desired correctness properties specified by the user. Advantageously, in accordance with the invention, the model extraction process and application of the model checker occurs in a direct and dynamic fashion from the subject event driven system program code without the need for user intervention.

**[0017]** The present invention can be embodied in the form of methods and apparatuses for practicing those methods. The invention can also be embodied in the form of program code embodied in tangible media, such as floppy diskettes, CD-ROMs, hard drives, or any other machine-readable storage medium, wherein, when the program code is loaded into and executed by a machine, such as a computer, the machine becomes an apparatus for practicing the invention. The invention can also be embodied in the form of program code, for example, in a storage medium, loaded into and/

or executed by a machine, or transmitted over some transmission medium, such as over electrical wiring or cabling, through fiber optics, or via electromagnetic radiation, wherein, when the program code is loaded into and executed by a machine, such as a computer, the machine becomes an apparatus for practicing the invention. When implemented on a general-purpose processor, the program code segments combine with the processor to provide a unique device that operates analogously to specific logic circuits.

[0018] In order to provide context and facilitate an understanding of the invention, an overview and discussion of an illustrative event driven system will now be presented. In particular, FIG. 1 shows a block diagram of an illustrative communications network which, as will be appreciated, contains a myriad of event driven applications. Communications network 100, e.g., is a public switched telephone network such as the well-known inter-exchange network of AT&T Corp., that provides long distance telephone services for its subscribers. These subscribers access communications network 100 through, e.g., communications devices 105-1 through 105-6, which are, e.g., customer premise equipment, wired telephones, personal computers, cellular telephones, pagers and facsimile machines. Communications network 100 includes, *inter alia*, a plurality of toll switches, e.g., shown in FIG. 1 as toll switches 115, 120, 125, and 130. These toll switches may be any of the well-known types of telecommunications switching equipment, e.g., the No. 4ESS® (Electronic Switching System) or the No. 5ESS® available from Lucent Technologies Inc., 600 Mountain Ave., Murray Hill, NJ 07974. As shown in FIG. 1, each of toll switches 115, 120, 125, and 130 are connected to a number of other switches via a so-called inter-toll network shown as block 150. Each toll switch may also be connected to multiple central offices ("CO"), e.g., CO's 110 through 114. The operation of such telecommunications networks and CO's is well-known, e.g., as discussed in "Engineering and Operations in the Bell System", Second Edition, Eighth Printing, International Standard Book Number 0-932764-04-5, 1993, and the detail of which will not be further discussed herein. In short, a CO is arranged to extend to a corresponding toll switch of communications network 100 a telephone call originating at, e.g., communications device 105-1, from which a calling party has dialed a particular telephone number. The CO, e.g., CO 110, is further arranged to extend the call connection to, e.g., communications device 105-6 associated with the called party and to the CO, e.g., CO 114, which receives the extension of the call from the corresponding toll switch, e.g., toll switch 125.

[0019] Toll switches 115, 120, 125 and 130 of communications network 100 are interconnected via data link 135, which may be, e.g., the well-known System Signaling 7 ("SS7") network. Communications network 100 is arranged so that the toll switches may exchange data messages with one another to establish a connection between a calling party (e.g., communications device 105-1) and a called party (e.g., communications device 105-6), via communications network 100. That is, the connection is made by extending a call through communications network 100 until the call is completed (e.g., the called party answers the call by going "off-hook") between the calling party and the called party. Communications network 100 further includes a number of centralized databases commonly known as Network Control Points ("NCPs"), a single one of which is shown as NCP 140. As is well-known, NCP's, such as NCP 140, are strategically positioned at various positions within communications network 100 to support various service features accessed and provided through the network such as the well-known "800" or "888" toll-free telephone number services.

[0020] As will be appreciated, communications network 100 and the various above-described elements of the network require a myriad of event driven applications. For example, call processing application programs for toll switches 115, 120, 125 and 130 must be developed to facilitate the extension of a call through network 100. In particular, for example, a calling party may wish to utilize communications device 105-1 for placing an outgoing call. Such an event driven call-processing application will include operations, i.e., control states, such as offhook, dial, idle, connect, ring, and busy, to name just a few.

[0021] As mentioned above, in accordance with the preferred embodiment of the invention, event driven software code is specified and written in accordance with the principles of the previously cited, co-pending, commonly assigned patent application, entitled "Method and Apparatus For Developing Event Driven Software" (hereinafter referred to as "the Method and Apparatus For Developing Event Driven Software application"). Thus, in order to further facilitate a complete understanding of the present invention, the various aspects of the invention in "the Method and Apparatus For Developing Event Driven Software application" will be now be discussed in some detail.

[0022] The various aspects of the invention in that application, provide a programming technique for defining, structuring, and developing event driven software. More particularly, event driven software is structured in two basic parts which are referred to therein and herein specifically as control-sensitive code and non-control-sensitive code. In accordance with the invention, control sensitive code includes the identification of control states, the events that are generated and that must be recognized, and the responses to the arrival of particular events in particular states. Further, within the control-sensitive code defining the system, control states are uniquely identified throughout such program code with a single symbol. The non-control-sensitive code, specified separately from the control sensitive code, is directed at the necessary normal sequential code for programming the event driven system such as the invocation of device drivers, the control of peripheral devices, the updating of tables, variable definition, etc. Advantageously, the invention allows the software developer to structure the event driven software application code such that a clear distinction is made between core control features, i.e., states, events and responses, and extraneous details not directly

impacting the behavior of the event driven system. As such, the programmer can determine the exact execution flow from control state-to-control state directly from a cursory examination of the program code itself as the individual control states have a strong visual context. That is, in accordance with the invention, the program source code is structured and defined such that the control states which define the event driven application have a direct and logical, i.e. causal, relationship which is immediately evident from an examination of the source code itself.

**[0023]** More particularly, FIG. 2 and FIG. 3 show an example of a conventional C source code program 200 (delimited by individual program code line numbers 1-268) directed to such an illustrative event driven application for processing an outgoing call. As will be appreciated, source code program 200 defines several control states for the application. See, for example, code sections 205 through 230 which define operations for the following respective control states: busy, conn, dead, digits, idle, and ring. Significantly, however, an examination of source code program 200 does not provide an immediately evident view of the execution thread between the various control states. From a high level programming perspective, there is no direct or logical, i.e. causal, relationship between the various control states defined within the source program code. That is, as mentioned above, in order for a programmer to understand the logical flow from control state-to-control state directly from the program source code it is necessary to trace the execution thread through the source, e.g., tracing branch instructions or "goto" statements. As will be appreciated, the tracing of such logical flow, particularly in very large source code programs, is a tedious, unilluminating, error prone, and time consuming task. For example, the source code for implementing all the requisite call processing features of communications network 100 will most likely be on the order of tens of thousands lines of code. This also leads to certain complexity in the testing of the software for so-called "bugs" and to determine whether the application program implements the desired features. Significantly, the novel programming technique is directed to defining, structuring, and coding event driven software which mitigates certain disadvantages of the prior art. In accordance with the invention, control sensitive code includes the identification of control states, the events that are generated and that must be recognized, and the responses to the arrival of particular events in particular states. Further, within the control-sensitive code defining the system, control states are uniquely identified throughout such program code with a single symbol. The non-control-sensitive code, specified separately from the control sensitive code, is directed at the necessary normal sequential code for programming the event driven system such as the invocation of device drivers, the control of peripheral devices, the updating of tables, variable definition, etc.

**[0024]** Advantageously, the invention allows the software developer to structure the event driven software application code such that a clear distinction is made between core control features, i.e., states, events and responses, and extraneous details not directly impacting the behavior of the event driven system. As such, the programmer can determine the exact execution flow from control state-to-control state directly from a cursory examination of the program code itself as the individual control states have a strong visual context. That is, in accordance with the invention, the program source code is structured and defined such that the control states which define the event driven application have a direct and logical, i.e. causal, relationship which is immediately evident from an examination of the source code itself.

**[0025]** In accordance with the preferred embodiment of the invention, a programming extension to the well-known ANSI-standard C programming language is adopted for facilitating the definition, structure and coding of event driven software in accordance with the invention. More particularly, in accordance with the preferred embodiment of the invention, the extension to the ANSI-standard C programming language consists of a single symbol, illustratively chosen to be the well-known ASCII symbol "@", and used for labeling all control states within the control sensitive portion of the program code. That is, the symbol "@" is used as the initial character of all labels that correspond to control states within the program source code. Thus, in accordance with the invention, a program statement prefixed with the "@" label identifies a control state in the program code whereby the current execution of the event driven system waits for the arrival of the next event. Following the arrival of the next event, execution of the subsequent program code follows along with the encoding of the required response to that event, and concluding with a transition to the new, uniquely defined control state. As will be appreciated, the response and the new control state, can be different for each event being processed depending upon operational issues such as any state information stored in internal tables, variables, device responses, and the like.

**[0026]** FIG. 4 shows a flowchart of illustrative operations for software development in accordance with the principles of the invention. More particularly, a specification for the event driven system to be programmed is defined (block 410), e.g., by the system engineer. As will be appreciated by those skilled in the art, such a specification provides a description of the overall application, e.g., event driven system, for which a software program is to be written and implemented. From the system specification, the control states for implementing the event driven system under development can be identified (block 420) which, in turn, leads to the identification of the events which will be generated and that must be recognized, and the responses to the arrival of particular events in particular states. As such, in accordance with the invention, the definition, structuring, and coding (block 430) of the source code program is divided into two parts, as discussed above, the control-sensitive code and the non-control-sensitive-code.

**[0027]** In accordance with the invention, within the control-sensitive code defining the system, control states are uniquely identified throughout such program code with a single symbol. As discussed above, in accordance with the

preferred embodiment of the invention, the symbol "@" is used as the initial character of all labels that correspond to control states within the control-sensitive portion of the overall program source code. Thus, in accordance with the invention, a program statement prefixed with the "@" label identifies a wait-state in the program code, where the current execution of the event driven system waits for the arrival of the next event. Further, the non-control-sensitive code, specified separately from the control sensitive code, is directed at the necessary normal sequential code for programming the event driven system such as the invocation of device drivers, the control of peripheral devices, the updating of tables, variable definition, etc.

**[0028]** Advantageously, the invention allows the software developer to structure the event driven software application code such that a clear distinction is made between core control features, i.e., states, events and responses, and extraneous details not directly impacting the behavior of the event driven system. Thus, one practical result of applying the principles of the invention is the programmer can determine the exact execution flow from control state-to-control state directly from a cursory examination of the program code itself as the individual control states have a strong visual context. That is, in accordance with the invention, the program source code is structured and defined such that the control states which define the event driven application have a direct and logical, i.e. causal, relationship which is immediately evident from an examination of the source code itself. Significantly, our present invention leads to further advantages in the testing and verifying of the event driven software (block 440) to determine if the software satisfies the desired properties as specified by the user. In particular, it is the testing of such event driven software that the present invention is directed and further discussed below in greater detail.

**[0029]** However, continuing with the discussion of "the Method and Apparatus For Developing Event Driven Software application", as mentioned above, that invention allows the software developer to structure the event driven software application code such that a clear distinction is made between core control features, i.e., states, events and responses, and extraneous details not directly impacting the behavior of the event driven system. For example, FIG.'s 5 and 6 show an illustrative C source code program 500 (delineated by individual program code line numbers 1-258), structured and defined in accordance with the invention, for the illustrative telecommunications call processing application of FIG.'s 2 and 3. In particular, control states 510 through 555 are delineated, in accordance with the preferred embodiment, with the symbol "@" as the initial character of all labels thereby designating the control states within the control-sensitive portion of the overall source code program 500. Thus, source code program 500, in accordance with the invention, has a direct and logical, i.e. causal, relationship between the individual control states, i.e., control states 510-555. Significantly, the programmer can determine the exact execution flow from control state-to-control state directly from a cursory examination of the program code itself as the individual control states have a strong visual context.

**[0030]** For example, through a cursory examination of source code program 500, one can quickly and exactly ascertain when viewing the code at control state 535, i.e., the "@digits" control state, how that particular control state is reached. That is, invocation and execution of this particular event driven system, i.e., the processing of an outgoing call, a control state for allowing the dialing of digits on a telephone device is processed. Control state 535 is such a state and is clearly identifiable beginning at line 125 of code 500, in accordance with preferred embodiment of the invention. In viewing control state 535, one can readily see that the execution flow in reaching this control state is directly through the above-coded control states 510 through 530, respectively. Further, if source code program 500 needs to be revised to add another control state, e.g., to add some further application-specific functionality, the new control state can be placed at the exact location within the existing control state hierarchy without losing context between all the control states.

**[0031]** This logical, i.e. causal, relationship among control states and the clear execution flow directly perceived from viewing source code program 500 is further evident in FIG.'s 7 and 8 which show a translated source code program 700 derived from the illustrative C source code program 500 of FIG.'s 5 and 6. As mentioned above, in accordance with the preferred embodiment of the invention, the extension to the ANSI-standard C programming language consists of a single symbol illustratively chosen to be the symbol "@" and used for labeling all control states within the control sensitive portion of the program code. The mechanics, i.e. the translator, of the actual translation of the code itself will be readily apparent to those skilled in the art. Thus, translated source code program 700 is the actual fully translated ANSI-standard C representation of the event driven application program in accordance with the invention. For example, control state 515 (i.e., "@idle") of FIG. 5 is translated to program code section 710, and control state 545 (i.e., "@ring") is translated to program code section 720.

**[0032]** The above-described advantages of the invention in "the Method and Apparatus For Developing Event Driven Software application" provide significant utility in the computer programming arts, in particular, to the definition, structure, and development of event driven software application programs. Of course, critical to the successful development of any application program is the effective testing and verification of the program code against a defined set of expected properties. It is the testing of event driven software that the various aspects of the present invention are directed and which now will be discussed in greater detail.

**[0033]** The various aspects of the present invention are directed to the efficient testing of event driven software and making it possible to track the validity of future changes to the program code, as part of the normal maintenance,

extension and revision process to the source code. FIG. 9 shows a flowchart of illustrative operations for testing event driven software in accordance with the principles of the invention. More particularly, in accordance with the preferred embodiment of the invention, event driven software code is specified and written in accordance with the principles of the invention in "the Method and Apparatus For Developing Event Driven Software application", as described previously.

That is, in accordance with the preferred embodiment, the source code program is defined and structured into control-sensitive and non-control-sensitive format (block 910.) In accordance with the invention, the event driven source code program is parsed to extract the control states defined in the source program, and to convert the source code into an intermediate state machine format (block 920.) Thereafter, in accordance with the invention, the intermediate state machine format is converted into an automata-based format for modeling checking purposes.

[0034] In particular, in accordance with the preferred embodiment, the intermediate state machine format is converted into an annotated automaton model in the input language of a specific model checker (block 940). In accordance with the preferred embodiment, the model checker tool is the well-known "SPIN" model checker developed by and available from the Bell Laboratories Division of Lucent Technologies Inc., and as described in more detail, e.g., in G. J. Holzmann, *The Model Checker SPIN, IEEE Trans. On Software Engineering*, Vol. 23, No. 5, pp. 279-295, May 1997, which is hereby incorporated by reference for all purposes. Importantly, in accordance with the invention, each statement from the source code program is mapped (block 930) onto an abstraction within the automaton model. The mapping, discussed below in greater detail, is facilitated by a translation map which is defined at the initial stage of the verification process. The map need not be revised thereafter unless new types of instructions are introduced, e.g., after revisions are made to repair programming faults, into the source program. The mapping, *inter alia*, dictates a fixed translation of the particular statement into the target language of the model checker. Thus, instructions that appear in the map are systematically converted into the automaton model wherever such instructions appear in the source code.

[0035] In addition to the above-described map, in accordance with the invention, a so-called environment model is defined, illustratively by a user, which encapsulates a minimal set of assumptions that must be made about the particular operating environment in which the event driven application is executed. More particularly, as the model checker input language is provided (block 950) to the model checker, e.g., SPIN, the environmental model is applied (block 975). As a result, in accordance with the preferred embodiment of the invention, the verification of the properties of the event driven system is made subject to the environmental model during the testing and checking (block 960) of the event driven software by the model checker. Thereafter, the testing and checking results are output, in a well-known manner, by the model checker (block 970) for determining whether the event driven system conforms with the user's desired execution properties and behaviors.

[0036] Important to the above-described testing and checking of event driven software, in accordance with the invention, is the efficient parsing of the event driven source code program to extract the control states defined in the source program, and converting the source code into an intermediate state machine format (see, e.g., FIG. 9, block 920.) That is, to perform model checking efficiently, one must be able to identify the control states, events and actions (i.e., responses) of the event driven system. Thus, a further aspect of the invention is directed to our realization of a formal specification which facilitates the description of an event driven system for model checking purposes, as will now be discussed.

[0037] In accordance with this aspect of the invention, a format is defined which facilitates the specification of an event driven system. In accordance with various embodiments of the invention (discussed further below), the specification may be converted into a target programming language, e.g. C., for direct compilation and execution. In accordance with still further embodiments of the invention, the specification is converted, using the above-described mapping, into a logical verification model such as the input language of the SPIN model checking tool. As will be appreciated by those skilled in the art, grammar specification tools are well-known. For example, the YACC (yet-another-compiler-compiler) is one well-known UNIX software utility tool (see, e.g., A. T. Schreiner, *Using C with curses, lex and yacc*, Prentice-Hall Inc., Englewood Cliffs, NJ 07632, 1990) which assists programmers in the development of C routines which analyze and interpret an input stream, and also facilitates the development of compilers and interpreters. Similar to YACC, our specification format provides a formalism for describing the grammar of a language, and automating the generation of a compiler for that language. However, our specification format, in accordance with the invention, is directed towards the specification of the behavior of the system.

[0038] More particularly, an illustrative grammar definition, in accordance with the preferred embodiment of the invention is as follows:



---

```

5      spec:  defs “%%” rules
           ;
      defs:  /*empty*/
10         |      one_def
           |      one_def defs
           ;
15      one_def:  “%c_template” “\” “ STRING ”\””
                |  “%p_template” “\” “ STRING ”\””
20         |  “%p_map” “\” “STRING ”\””
                |  “%nparam”NUMBER
                |  “%event” names
25         |  “%state” names
           ;

30      names:  STRING
                |  STRING names
           ;
35

      rules: one_rule
                |  one_rule rules
40         ;

45      one_rule:  source “:” opt_recpt responses “,”
                ;

50      responses:  one_response
                |  one_response “|” responses
      one_response:  triggers opt_target opt_action
55         ;

```

```

triggers:      one_trigger
               |      one_trigger "+" triggers
5
               ;

one_trigger:event
10
               |      "("condition")"
               |      "!("condition")"
               |      "else"
15
               |      "always"
               ;

opt_target:    /*empty*/
20
               |      target
               ;

opt_action     /*empty*/
25
               |      "{"action"}"
               ;

opt_recpt:     /*empty*/
30
               |      onreceipt{"action"}"
               ;
35

event:         STRING;
condition:     STRING;
40
action:        STRING;
source:        STRING;
45
target:        STRING | "@" STRING;

```

---

50 The above illustrative grammar definition provides the framework for defining a specification for a particular application, e.g., an event driven system, in accordance with the principles of the invention. All items in the illustrative grammar definition which appear in double quotes are terminal symbols (keywords, literals, tokens). The capitalized word STRING represents an alphanumeric string, and NUMBER represents a sequence of one or more digits. Alternatives are separated by a vertical bar "|", i.e., pipe symbol, and each grammar rule is terminated by a semicolon.

55 **[0039]** In accordance with the preferred embodiment, the specification format consists of two parts: a declaration and a specification. The declaration part of the specification of the preferred embodiment, contains, *inter alia*, a listing of symbolic names for all events that will be used in the specification. For example, the following code fragment is an

illustrative declaration in accordance with the preferred embodiment:

---

```

5
    %c_template      "c.ctp" /*optional: a C code template */
    %p_template      "p.ctp" /*optional: a SPIN model template */
10    %p_map          "p.map" /* optional: maps C code to SPIN */

    %event Cranswer
    %event Crbusy
    %event Crconn
15    %event Crdigit

    %state S_idle
    %state S_pots
    %state S_alloc
    %state S_parked
20    %state S_busy
    %%
35

```

---

40 In the above illustrative declaration, all words that begin with a percent "%" symbol are keywords. For example, "%c\_template" defines the name of a file that contains a template for the final implementation of the code in C, i.e., the target implementation language of the preferred embodiment. Further, the "%p\_map" statement gives the name of a file that can be used to define a relation between the C code from the implementation and the abstract representations of the code, i.e., mapping, used for verification with the model checker. In addition, the illustrative declaration above further contains a listing of symbolic names for all events, e.g., "%event Cranswer", that will be used in the specification, as well as, a list of state names for all of the control states, e.g., "%state S\_idle", in the specification.

45 **[0040]** The core of the specification format, in accordance with the invention, is the specification of a set of transition rules for the event driven system wherein the state specification consists of a sequence of transition rules specified in the following notation:

```

50      state   : event1 targetstate1 {code fragment1}
               | event2 targetstate2 {code fragment2}
55      .....etc.
               ;

```

For example, the following code fragment is an illustrative control state specification in accordance with the preferred embodiment:

```

5      _____
      |
      |
      | S_idle : Cronhook S_idle
10      | {x→cwswit &= Swhook;}
      | | Crflash
      | {x→cwswit |= Swhook;}
15      | | Croffhook @S_idle_E3
      | {x→cwswit |= Swhook;}
20      | | Iring @S_idle_E5
      | ;
      |
      |_____

```

25

In the above illustrative control state specification, the transition rule begins with the name of the state, e.g., "S\_idle", followed by a colon. After the colon, a series of possible transitions out of this state are listed. The individual transitions are separated from each other by a vertical bar "|". Each complete transition has three parts: the name of an event that triggers the response, the name of a new state that is reached after the response has been performed, i.e., "the next-state", and the response itself. The next-state part of the specification defines where control moves after the response has been completely processed, and contains the name of either a predefined control state or a so-called internal state. If two events have the same next-state and response, in accordance with further embodiments of the invention, an abbreviated notation can be used such as:

```

40      |_____
      |_____
      | | Crflash
      | + Crdis
45      | + Croffhook S_idle
      | {x→cwsit |= Swhook;}
      |_____
50      |_____

```

55

Using the above noted abbreviated notation, the events Crflash, Crdis, and Croffhook will all generate the same response, i.e., the execution of the specified code fragment and a transition to the state "S\_idle".

**[0041]** As will be appreciated, the event driven system is meant to stop execution at each control state and wait for the arrival of the next event. In contrast, the event driven system passes through internal states without waiting. In addition, state names in the next-state segment of a transition rule can be prefixed with the "@" symbol (it should be noted that this particular notation is not the same as that described previously with regard to identifying control states

in accordance with the various aspects of the invention in "the Method and Apparatus For Developing Event Driven Software application") to indicate that processing of the next state should continue without waiting for a next event to arrive. In such a case, the last event remains available if an event is needed to determine the behavior of the system at a next state.

**[0042]** With reference to the internal states mentioned above, such events encode an internal, non-event related decision of the event driven system. In accordance with the preferred embodiment of the invention, the syntax in the specification for internal events is the same as for control states, except that instead of event names, general Boolean conditions in the target implementation language are used. For example, an illustrative internal state is defined as follows:

---

```

S_idle_E5:    {x→drv→trunk} @S_otrunk
              {x→orig=ipc→orig;
              ipc→orig→term = x;
              emitcdr{x,CDRtransit,0};}
              || {x→drv→trunk} @S_idle_E6
              ;

```

---

The above illustrative internal state specifies that the value of a specific field (in this case hiding two levels inside a C data structure named "x") will decide whether control moves into state "S\_otrunk" or state "S\_idle\_E6". In the first case, the piece of action code defined after the state will be executed. In the latter case, no code is executed.

**[0043]** In accordance with the preferred embodiment, if the name of an event or condition is "else" this indicates that a default response will be taken when none of the explicitly listed event names or Boolean conditions apply. For example, the above illustrative internal state code fragment can also be written as follows:

---

```

S_idle_E5:    {x→drv→trunk} @S_otrunk
              {x→orig=ipc→orig;
              ipc→orig→term = x;
              emitcdr{x,CDRtransit,0};}
              | else @S_idle_E6
              ;

```

---

In accordance with this alternative internal state code fragment, no code fragment is executed when the condition "{x→drv→trunk}" evaluates to "false" upon reaching this state. Instead, only a transition into state "S\_idle\_E6" will result.

**[0044]** Further, with regard to the specification of the event driven system, it is sometimes useful to define a common action fragment that must be executed immediately following the reception of an event in a control state. By definition,

such a code fragment is to be executed before any of the transition rules are applied. In accordance with the preferred embodiment of the invention, the keyword "onreceipt" is employed to define such common action fragments. An illustrative example of this keyword's use is as follows:

```

5
_____
_____
10      S_orig : onreceipt
           {x→cwswitch&= Swconn;
           y=callroute(x→endpt→ctxt, x→lidx→number,&i);}
15      | (y == nil) @S_orig_E54
           | else @S_orig_E55
           ;
20
_____
_____

```

25 In the above illustrative code fragment, the state is an internal state so it can always be referred to in a next-state segment with the "@" prefix. The "onreceipt" code is immediately executed when reaching this state on a transition, and the condition "(y == nil)" is evaluated. If the condition is "true", the system moves into another internal state named "S\_orig\_E54", and if "false", the system move into internal state "S\_orig\_E55" where processing continues without waiting for a new event.

30 **[0045]** Finally, with regard to the specification, it is sometimes useful to bypass condition checks altogether and specify an intermediate transition state that will lead unconditionally to a successor state. In accordance with the preferred embodiment of the invention, a condition that is always true is represented by the keyword "always" as is illustrated in the following example code fragment:

```

35
_____
_____
40      S_enabdial : onreceipt
           {x→ndigits=0;
           x→drv→dtmf(x,l);
45      x→cwswit |= Swconn;}
           | always S_dial
50      ;
_____
_____
55

```

**[0046]** Our realization of the above-described formal specification in describing event driven systems for model checking purposes advantageously provides a specification format for describing the behavior of an event driven system.

[0047] To further illustrate the various aspects and advantages of the invention in testing event driven software, FIG. 10 shows intermediate state machine code 1000 (delineated by individual program code line numbers 1-124) derived, in accordance with the principles of the invention as discussed above, from the source code listing of the illustrative C source code program 500 of FIG.'s 5 and 6. As discussed above, to effectively perform model checking on event driven code, an accurate assessment and identification of the control states, events, and actions must occur. Advantageously, intermediate state machine code 1000, in accordance with the principles of the invention, provides a clear delineation amongst control states, events and actions. More particularly, in accordance with the preferred embodiment, a translator is applied to C source code program 500 to directly generate intermediate state machine code 1000. The actual construction of the translator will be readily understood by those skilled in the computer programming art and need not be discussed further herein. However, by structuring C source code program 500 in accordance with the present invention, the automatic identification of the relevant control states, events, and actions is made possible. Thus, the advantages of the present invention as set forth herein are realized.

[0048] More particularly, a careful examination of intermediate state machine code 1000 show a clear delineation of events (see, e.g., code section 1010) and control states (see, e.g., code section 1020) of the declaration part of code 1000. Further, the clear interaction amongst and between such events, control states and the related actions is shown, in accordance with the invention, by the various state specifications (see, e.g., state specifications 1030 through 1050) within intermediate state machine code 1000. As discussed above with reference to the operations of FIG. 9, in accordance with the invention, intermediate state machine code 1000 is converted to the model checker input language for executing the testing of the subject program code. This conversion is facilitated by a translation map which is defined at the initial stage of the verification process. The mapping, *inter alia*, dictates a fixed translation of the particular statement into the target language of the model checker. Thus, instructions that appear in the map are systematically converted into the automaton model wherever such instructions appear in the original source code.

[0049] FIG. 11 shows an illustrative map 1100 for the event driven system defined by the illustrative C source code program 500. Map 1100 consists of program code 1110 (delineated by individual program code line numbers 1-47) which defines how the fixed translation of particular statements in the original source code is made into the target language of the model checker. In particular, program code 1110 includes event definitions 1120 and code definitions 1130. For example, line 20 of program code 1110 maps every occurrence of the statement in the left hand column onto the abstraction given in the right hand column. In accordance with the invention, the application of map 1100 results in the derivation of model checker input language 1200 (delineated by individual program code line numbers 1-303) shown in FIG.'s 12 and 13. Further, in accordance with the invention, a set of library functions is generated, illustratively shown as library functions 1400 in FIG. 14, for the model checker input language of FIG.'s 12 and 13. As will be appreciated, library functions 1400 (delineated by individual program code line numbers 1-153) provide the necessary semantics for defining terms in model checker input language 1200. In accordance with the preferred embodiment, model checker input language 1200 is directed to and useful with the SPIN model checker tool.

[0050] Significantly, one important feature of using maps, in accordance with the invention, is the capability provided to the user in defining precisely the appropriate level of detail at which the checking process will be applied, in a source independent manner. The map can contain "true" or "skip" as the translation (i.e., right hand column of program code 1110 described above) of any statement which indicates that this detail is abstracted away. Therefore, the map acts as a "filter" to filter out extraneous detail, in addition to, acting as a converter from the target programming language, e.g., C, to the verification model. The map also serves as a complete formalization of the linkage between source code and the verification model.

[0051] As described above, in accordance with the preferred embodiment of the invention, the verification of the properties of the event driven system is made subject to an environmental model during the testing and checking (see, FIG. 9, block 975) of the event driven software by the verification system. The environmental model formalizes particular assumptions, made by the user, about the environment in which the event driven program code is to be executed. For example, in the case of the illustrative telecommunications call processing application described herein, the environmental model might include formalizations directed to subscriber behavior, hardware responses, and communications network performance. In particular, FIG. 15 shows an illustrative environmental model 1500 (delineated by individual program code line numbers 1-80) for the telecommunications processing application described in FIG.'s 5 and 6. For example, code fragment 1510 of environmental model 1500 describes a formalization, i.e. a template, for a telecommunications device useful in the illustrative telecommunications application. In accordance with the preferred embodiment, the model checker, i.e., SPIN, will use model checker input language 1200 in conjunction with environmental model 1500 to determine whether the event driven system conforms with the user's desired execution properties and behaviors.

[0052] Advantageously, the separate definition of the map and environmental model, in accordance with the invention, ensure that little, if anything, in the verification infrastructure needs to be updated when routine changes are made in the original source code. That is, unless new basic types of statements are introduced in the source code, the verification process can be repeated without user intervention and the source code may be checked for continued

compliance against a large library of essential correctness properties. Further, in accordance with the invention, a precise automaton model representative of the event driven code under test is automatically generated directly from the source code and verified with a logic model checker, e.g., SPIN.

[0053] For example, FIG. 16 shows an illustrative automaton model 1600 for a particular control state defined by the telecommunications processing application described in FIG.'s 5 and 6. More particularly, the control state depicted in FIG. 16 is the so-called "ring" control state defined in source code program 500 as control state 545 (see, FIG.'s 5 and 6). The "@ring" program statement in line 183 of source code program 500 begins the clear delineation of this particular control state from a source program viewpoint. In accordance with the testing and checking aspects of the invention, as described above, from this control state a precise automaton, e.g., automaton model 1600, is extracted. Automaton model 1600 clearly shows the transitions in and out of the "ring" control state 1610 and amongst other control states of the application, i.e., "digits\_1" control state 1620, "idle" control state 1630, "conn" control state 1640, and "error" control state 1650. Automaton model 1600 also clearly illustrates the various events, i.e., event 1660 through event 1695, occurring in the illustrative application. Further, full automaton model 1700, shown in FIG. 17, is an illustrative full automaton model of the complete telecommunications processing application described in FIG.'s 5 and 6, and automatically generated in accordance with the invention.

[0054] The above-described embodiments of the invention employ a C source program written (see, e.g., FIG.'s 5 and 6) in accordance with the principles of the "the Method and Apparatus For Developing Event Driven Software application", and tested in accordance with the principles of the present invention as described above. We have further realized that certain types of users, e.g. verification system programmers or software testing engineers, may wish to begin their system design process by defining the event driven in accordance with the specification format described above. For example, such users may employ the specification format set forth herein to directly code the subject event driven system. Thus, in accordance with further embodiments of the invention, the specification is converted into a target programming language, e.g. C., for direct compilation and execution. In accordance with such further embodiments of the invention, a template is employed to facilitate such a conversion. The following is an illustrative C code template for use in converting this such state machine code to standard C for compilation and execution:

---

```

void
state(Line *x, Ipc *ipc)
{
    Line *y;
    int op, i, c, os, time;

    goto start;

```



```

    /*** template: ***/
5      @C

    out:
10      if (x == nil)
        return;
        if (time != 0)
15          {
            x->time=rtms()+time;
20          addtimer(x);
          }
        return; /* exit-point */
25      start:
        op=ipc->type; os=x->state; time=0; switch (os)
        {
30          @@
        }
        os=0;
35      error:
        y=x->term; x->term=nil; goto Aerror;
40  }

```

---

45 [0055] The above-described C code template provides the framework for a complete conversion of the state machine specification to C. As will be appreciated, the illustrative C code template can be executed by the model checker tool itself or on a separate machine. The template includes type definitions for the arguments that are passed to a routine called "state" that is used as the core of the conversion implementation. The routine must be called whenever an event occurs that relates to the subject event driven system. The template code further contains a place-holder "@C" for the state machine that is generated from the transition rules specified in accordance with the invention. The actual program code necessary for the generation of the state machine will be readily understood by those skilled in the art and need not be further detailed herein. At each call, the "start" routine executes and retrieves an event type and parameter value from the data structure that is used by the environment to store detailed information about the current event. The current state of the process is retrieved from the data structure in the first argument to the routine, and a so-called "C switch" is performed on this state value to establish the proper position in the transition rules. The short-hand symbol "@@" of the template is expanded by the tool into a full list of names of all control states, with matching jumps to the places in the generated, i.e., converted, C code where the code fragments for the transition rules of each state are

placed. Advantageously, in accordance with these further embodiments of the invention, the specification is converted into a target programming language, e.g., C, for direct compilation and execution.

[0056] For example, FIG. 18 shows an illustrative computer system 1800 useful in executing and testing event driven software code in accordance with the present invention. In particular, computer system 1800 includes conventional personal computer 1810 which is connected to display 1820 and keyboard 1830. As will be appreciated, information produced by personal computer 1810, e.g., during execution of particular event driven software code, may be displayed on display 1820 for access by a user of computer system 1800. The user, as a result of viewing display 1820, interacts with and controls personal computer 1810, in a conventional manner, using keyboard 1830. Further, mouse 1880, or other well-known pointing devices, may allow the user to provide additional inputs to personal computer 1810. Illustrative computer system 1800 further includes processor 1840 which, illustratively, executes and tests event driven application software code in accordance with the present invention. For example, such event driven application software code may be loaded into personal computer 1810 via conventional diskette 1870 and thereafter stored in internal disk memory 1860. Further, execution and testing of the code may require access and use of random access memory 1850 ("RAM") in a conventional manner. Thus, the user of computer system 1800 can execute and test software, in accordance with the present invention, to provide a wide variety of applications, e.g., an event driven system. Further, as will be appreciated, computer system 1800 may also be connected, in a conventional manner, to other computer systems or devices for execution and testing of the particular application programs.

[0057] For example, computer system 1800 can be used to execute and test software, in accordance with the present invention, resulting in error trace 1900 as shown in FIG. 19. Error trace 1900 consists of program code 1910 (delineated by individual program code line numbers 1-47) and is an illustrative error trace, generated in accordance with the principles of the invention, for the telecommunications processing application described in FIG.'s 5 and 6. More particularly, the illustrative error trace 1900 shows how the checking process produces a scenario which directly identifies the presence of errors in the original source code of the event driven system. Further, error trace 1900 through program code 1910 identifies such errors in terms of programming level statements, e.g., C-level statements, that may be executed to reproduce the error. In the illustrative error trace 1900, the trace demonstrates in a direct sequence of C-program code statement (see, FIG. 9, program code 1920) executions that leads the event driven system into the illustrate state "@busy" with "Crdigit" being the first event to be processed. As can be seen and appreciated from error trace 1900, in conjunction with, C source program code 500 (in particular, control state 555 at line 223), there is no provision in the program code, i.e., a coding error, to anticipate the possible occurrence of this event. Advantageously, in accordance with the invention, a user can interpret error trace 1900 solely in terms of C-level execution statements without any specialized knowledge of the model itself or significant parts of the intermediate representations that are used in the checking process itself.

[0058] The foregoing merely illustrates the principles of the present invention. It will thus be appreciated that those skilled in the art will be able to devise various arrangements which, although not explicitly described or shown herein, embody the principles of the invention and are included within its scope. Furthermore, all examples and conditional language recited herein are principally intended expressly to be only for pedagogical purposes to aid the reader in understanding the principles of the invention and the concepts contributed by the Applicant(s) to furthering the art, and are to be construed as being without limitation to such specifically recited examples and conditions. Moreover, all statements herein reciting principles, aspects, and embodiments of the invention, as well as specific examples thereof, are intended to encompass both structural and functional equivalents thereof. Additionally, it is intended that such equivalents include both currently known equivalents as well as equivalents developed in the future, i.e., any elements developed that perform the same function, regardless of structure.

[0059] Thus, for example, it will be appreciated by those skilled in the art that the block diagrams herein represent conceptual views of illustrative circuitry embodying the principles of the invention. Similarly, it will be appreciated that any flowcharts, flow diagrams, state transition diagrams, pseudocode, program code, and the like represent various processes which may be substantially represented in computer readable medium and so executed by a computer, machine, or processor, whether or not such computer, machine, or processor, is explicitly shown.

[0060] Further, in the claims hereof any element expressed as a means for performing a specified function is intended to encompass any way of performing that function, including, for example, a) a combination of circuit elements which performs that function; or b) software in any form (including, therefore, firmware, object code, microcode or the like) combined with appropriate circuitry for executing that software to perform the function.

## Claims

1. A software testing method, carried out on a source program of the software, the source program including a plurality of instructions, said method comprising:

parsing the source program to identify a plurality of control states and converting the source program to a state machine format;  
 applying a translation map to the plurality of instructions of the source program;  
 converting the state machine format of the source program to a verification program;  
 5 applying an environmental model to the verification program; and  
 testing the software in a verification tool using the verification program.

2. The software testing method of claim 1 wherein the source program contains a first source code segment for causing a computer to execute an event driven system; the first source code segment including a first plurality of instructions defining the plurality of control states, the plurality of control states being associated with the event driven system, a second plurality of instructions defining a plurality of events of the event driven system, and a third plurality of instructions defining a plurality of actions of the event driven system, each control state of the plurality of control states having an individual identifier within the first plurality of instructions of the first source code segment, the individual identifier being a programming symbol which is common to all the identified control states, and a second source code segment containing at least one plurality of instructions defining a sequential control of the computer executing the event driven system.

3. The software testing method of claim 1 or claim 2 wherein the translation map includes a plurality of mapping instructions.

4. The software testing method of claim 3 wherein the applying the translation map operation includes:

comparing the plurality of mapping instructions to the plurality of instructions of the source program to detect a match; and  
 converting particular ones of the mapping instructions which match particular ones of the instructions of the source program to the input language of the model checker.

5. The software testing method of any of the preceding claims wherein the environmental model includes a plurality of operational attributes of the event driven system.

6. The software testing method of any of the preceding claims wherein the verification tool is a model checker tool.

7. The software testing method of claim 6 wherein the model checker tool is a SPIN model checker tool.

8. The software testing method of any of the preceding claims wherein the event driven system is a telecommunications service application, the telecommunications service application having at least one event associated with extending a telephone call through a communications network.

9. The software testing method of any of the preceding claims wherein the state machine format is converted using a specification defined by a user.

10. The software testing method of any of the preceding claims further comprising:

compiling the first source code segment and the second source code segment into an object program; and  
 executing the object program to provide the event driven system.

11. The software testing method of any of the preceding claims wherein the step of testing uses at least one set of properties defined by a user associated with the execution of the event driven system.

12. The software testing method of claim 11 wherein the translation map is specified by the user.

13. The software testing method of claim 12 wherein the translation map, the environmental model, and the series of programming statements are each written in the C programming language.

14. The software testing method of any of the preceding claims including the step of displaying a result of the testing of the software.

15. The software testing method of claim 14 wherein the displayed result includes at least one full automaton repre-

sentation of the event driven system.

16. A software testing apparatus comprising means arranged to carry out each step of a method as claimed in any of the preceding claims.

17. A machine-readable medium having stored thereon a plurality of instructions, the plurality of instructions including instructions that, when executed by a machine, cause the machine to perform a method as claimed in any of claims 1 to 15.

18. A software testing system comprising:

a computer having at least one memory;

means for receiving a source program of software to be tested and comprising series of programming language statements in the memory;

and means arranged to carry out each step of a method as claimed in any of claims 1 to 15 to test said software.

FIG. 1

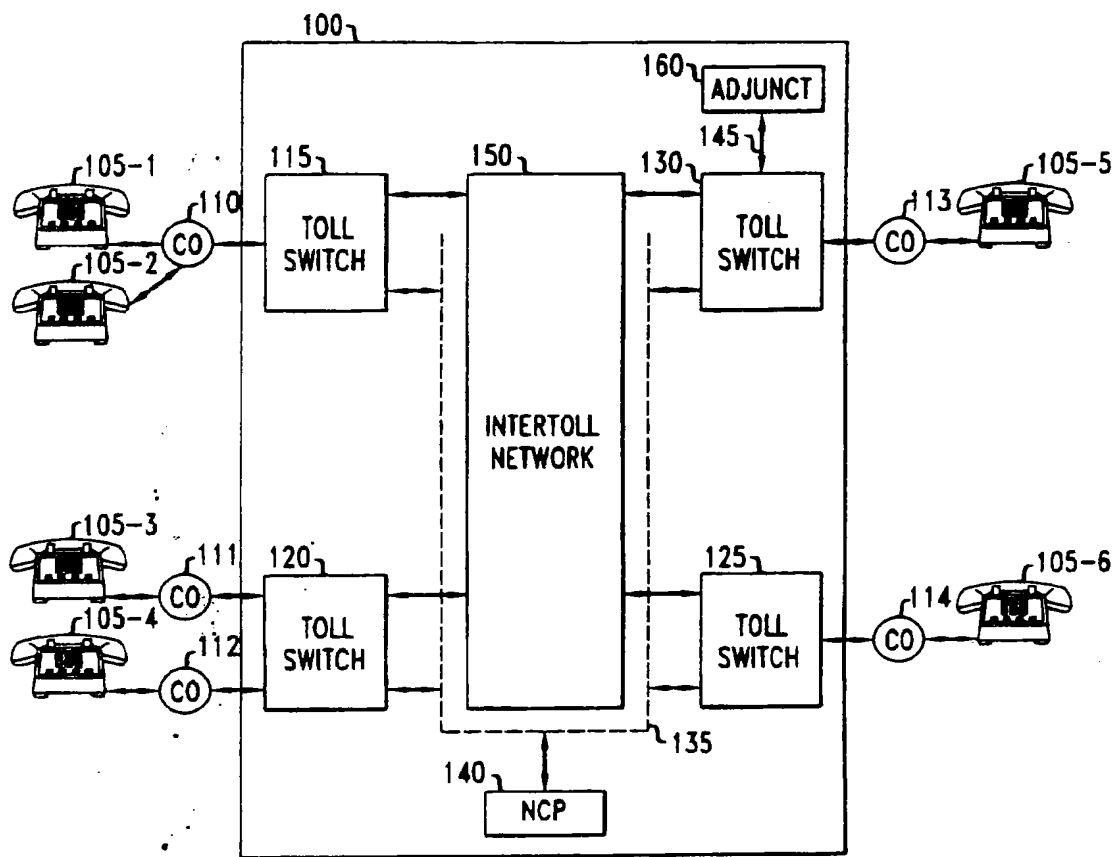


FIG. 2

```

1 //equivalent of state.z in traditional notation
2 //first switch on current state, then on event
.
.
.
69 default:
70 print ("no such state %d, line %d\n", x->lineno);
71 break;
72
73 case busy: //phone is connected to a tone until hung up
74 switch(op) {
75 default:
76 goto error;
77
78 case Crtone:
79 x->state = busy;
80 break;
81
82 case Cronhook:
83 send(x, Cldis, 0);
84 x->state = idle;
85 break;
86 }
87 break;
.
.
.
89 case conn: //phones are connected
90 switch(op) {
91 default:
92 case Crconn: //successful connection
93 break; //no state change
94
95 case Cronhook:
96 send(x->term, Cldis, 0);
97 send(x, Cldis, 0);
98 x->state = idle;
99 break;
100 }
101 break;
.
.
.
103 case dead; //phone unassigned
104
105 switch(op) { //check the event
106 default:
107 print("%l dead and op=%s\n", x, istring(op));
108 break; //no state change
109
110 case Crprovc: //line provisioned
111 x->term = 0; //initialize
112 x->state = idle; //next state
113 break;
114 }
115 break;
.
.
.

```

200

210

215

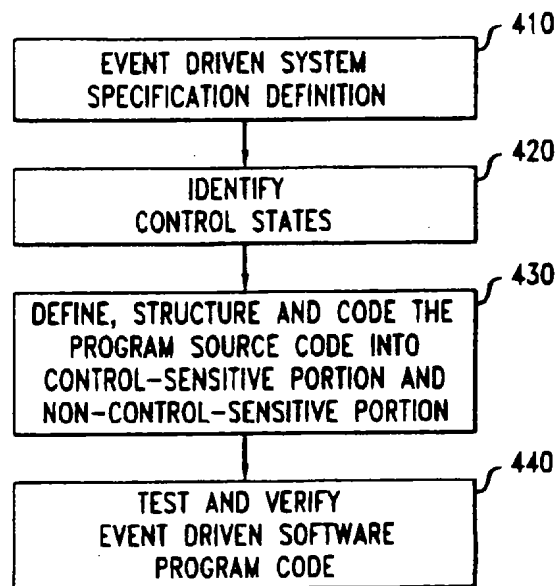
TO FIG 3

FIG. 3

```

116 FROM FIG 2
117 case digits:
118   x->time = seconds(6);
119   switch(op) {
120     default:
121       goto error;
122   }
123   case Cronhook: //line was hung up
124     send(x, Cldis, 0);
125     x->state = idle;
126     break;
127   case Crnone: //noise (dial tone or silence has been applied)
128     break;
129     //no state change
130   case Crdigit: //next digit detected
131     //decode digits according to dial plan
132     p = dialplan(x->digits, x->ndigits);
133     if(x->ndigits == 1) //first digit
134       send(x, Crnone, 0); //remove dial tone
135     if(p == 0) //incomplete
136       break;
137     //no state change
138   // translate local number to local phone
139   x->term = lookup(p);
140   if(x->term == 0) {
141     send(x, Crnone, Treadorder);
142     x->state = busy;
143   } else {
144     send(x->term, Ciring, 1); //ring destination
145     x->state = digits_2;
146   }
147   break;
148 }
149 case Crtime:
150   send(x, Cldis, 0);
151   send(x, Crnone, Tscram);
152   //one phone has audible ringing, other phone
153   // is ringing
154   switch(op) {
155     default:
156       goto error;
157   }
158   case Cronhook:
159     send(x->term, Cldis, 0);
160     send(x, Cldis, 0);
161     x->state = idle;
162     break;
163   case Crnone:
164     x->state = ring;
165     //no state change
166   }
167   //one phone has audible ringing, other phone
168   // is ringing
169   switch(op) {
170     default:
171       goto error;
172   }
173   case Cronhook:
174     send(x->term, Cldis, 0);
175     send(x, Cldis, 0);
176     x->state = idle;
177     break;
178   case Crnone:
179     x->state = ring;
180     //no state change
181   }
182   //one phone has audible ringing, other phone
183   // is ringing
184   switch(op) {
185     default:
186       goto error;
187   }
188   case Cronhook:
189     send(x->term, Cldis, 0);
190     send(x, Cldis, 0);
191     x->state = idle;
192     break;
193   case Crnone:
194     x->state = ring;
195     //no state change
196   }
197   //one phone has audible ringing, other phone
198   // is ringing
199   switch(op) {
200     default:
201       goto error;
202   }
203   case Cronhook:
204     send(x->term, Cldis, 0);
205     send(x, Cldis, 0);
206     x->state = idle;
207     break;
208   case Crnone:
209     x->state = ring;
210     //no state change
211   }
212   //one phone has audible ringing, other phone
213   // is ringing
214   switch(op) {
215     default:
216       goto error;
217   }
218   case Cronhook:
219     send(x->term, Cldis, 0);
220     send(x, Cldis, 0);
221     x->state = idle;
222     break;
223   case Crnone:
224     x->state = ring;
225     //no state change
226   }
227   //one phone has audible ringing, other phone
228   // is ringing
229   switch(op) {
230     default:
231       goto error;
232   }
233   case Cronhook:
234     send(x->term, Cldis, 0);
235     send(x, Cldis, 0);
236     x->state = idle;
237     break;
238   case Crnone:
239     x->state = ring;
240     //no state change
241   }
242   //one phone has audible ringing, other phone
243   // is ringing
244   switch(op) {
245     default:
246       goto error;
247   }
248   case Cronhook:
249     send(x->term, Cldis, 0);
250     send(x, Cldis, 0);
251     x->state = idle;
252     break;
253   case Crnone:
254     x->state = ring;
255     //no state change
256   }
257   //one phone has audible ringing, other phone
258   // is ringing
259   switch(op) {
260     default:
261       goto error;
262   }
263   case Cronhook:
264     send(x->term, Cldis, 0);
265     send(x, Cldis, 0);
266     x->state = idle;
267     break;
268   case Crnone:
269     x->state = ring;
270     //no state change
271   }
272   //one phone has audible ringing, other phone
273   // is ringing
274   switch(op) {
275     default:
276       goto error;
277   }
278   case Cronhook:
279     send(x->term, Cldis, 0);
280     send(x, Cldis, 0);
281     x->state = idle;
282     break;
283   case Crnone:
284     x->state = ring;
285     //no state change
286   }
287   //one phone has audible ringing, other phone
288   // is ringing
289   switch(op) {
290     default:
291       goto error;
292   }
293   case Cronhook:
294     send(x->term, Cldis, 0);
295     send(x, Cldis, 0);
296     x->state = idle;
297     break;
298   case Crnone:
299     x->state = ring;
300     //no state change
301   }
302   //one phone has audible ringing, other phone
303   // is ringing
304   switch(op) {
305     default:
306       goto error;
307   }
308   case Cronhook:
309     send(x->term, Cldis, 0);
310     send(x, Cldis, 0);
311     x->state = idle;
312     break;
313   case Crnone:
314     x->state = ring;
315     //no state change
316   }
317   //one phone has audible ringing, other phone
318   // is ringing
319   switch(op) {
320     default:
321       goto error;
322   }
323   case Cronhook:
324     send(x->term, Cldis, 0);
325     send(x, Cldis, 0);
326     x->state = idle;
327     break;
328   case Crnone:
329     x->state = ring;
330     //no state change
331   }
332   //one phone has audible ringing, other phone
333   // is ringing
334   switch(op) {
335     default:
336       goto error;
337   }
338   case Cronhook:
339     send(x->term, Cldis, 0);
340     send(x, Cldis, 0);
341     x->state = idle;
342     break;
343   case Crnone:
344     x->state = ring;
345     //no state change
346   }
347   //one phone has audible ringing, other phone
348   // is ringing
349   switch(op) {
350     default:
351       goto error;
352   }
353   case Cronhook:
354     send(x->term, Cldis, 0);
355     send(x, Cldis, 0);
356     x->state = idle;
357     break;
358   case Crnone:
359     x->state = ring;
360     //no state change
361   }
362   //one phone has audible ringing, other phone
363   // is ringing
364   switch(op) {
365     default:
366       goto error;
367   }
368   case Cronhook:
369     send(x->term, Cldis, 0);
370     send(x, Cldis, 0);
371     x->state = idle;
372     break;
373   case Crnone:
374     x->state = ring;
375     //no state change
376   }
377   //one phone has audible ringing, other phone
378   // is ringing
379   switch(op) {
380     default:
381       goto error;
382   }
383   case Cronhook:
384     send(x->term, Cldis, 0);
385     send(x, Cldis, 0);
386     x->state = idle;
387     break;
388   case Crnone:
389     x->state = ring;
390     //no state change
391   }
392   //one phone has audible ringing, other phone
393   // is ringing
394   switch(op) {
395     default:
396       goto error;
397   }
398   case Cronhook:
399     send(x->term, Cldis, 0);
400     send(x, Cldis, 0);
401     x->state = idle;
402     break;
403   case Crnone:
404     x->state = ring;
405     //no state change
406   }
407   //one phone has audible ringing, other phone
408   // is ringing
409   switch(op) {
410     default:
411       goto error;
412   }
413   case Cronhook:
414     send(x->term, Cldis, 0);
415     send(x, Cldis, 0);
416     x->state = idle;
417     break;
418   case Crnone:
419     x->state = ring;
420     //no state change
421   }
422   //one phone has audible ringing, other phone
423   // is ringing
424   switch(op) {
425     default:
426       goto error;
427   }
428   case Cronhook:
429     send(x->term, Cldis, 0);
430     send(x, Cldis, 0);
431     x->state = idle;
432     break;
433   case Crnone:
434     x->state = ring;
435     //no state change
436   }
437   //one phone has audible ringing, other phone
438   // is ringing
439   switch(op) {
440     default:
441       goto error;
442   }
443   case Cronhook:
444     send(x->term, Cldis, 0);
445     send(x, Cldis, 0);
446     x->state = idle;
447     break;
448   case Crnone:
449     x->state = ring;
450     //no state change
451   }
452   //one phone has audible ringing, other phone
453   // is ringing
454   switch(op) {
455     default:
456       goto error;
457   }
458   case Cronhook:
459     send(x->term, Cldis, 0);
460     send(x, Cldis, 0);
461     x->state = idle;
462     break;
463   case Crnone:
464     x->state = ring;
465     //no state change
466   }
467   //one phone has audible ringing, other phone
468   // is ringing
469   switch(op) {
470     default:
471       goto error;
472   }
473   case Cronhook:
474     send(x->term, Cldis, 0);
475     send(x, Cldis, 0);
476     x->state = idle;
477     break;
478   case Crnone:
479     x->state = ring;
480     //no state change
481   }
482   //one phone has audible ringing, other phone
483   // is ringing
484   switch(op) {
485     default:
486       goto error;
487   }
488   case Cronhook:
489     send(x->term, Cldis, 0);
490     send(x, Cldis, 0);
491     x->state = idle;
492     break;
493   case Crnone:
494     x->state = ring;
495     //no state change
496   }
497   //one phone has audible ringing, other phone
498   // is ringing
499   switch(op) {
500     default:
501       goto error;
502   }
503   case Cronhook:
504     send(x->term, Cldis, 0);
505     send(x, Cldis, 0);
506     x->state = idle;
507     break;
508   case Crnone:
509     x->state = ring;
510     //no state change
511   }
512   //one phone has audible ringing, other phone
513   // is ringing
514   switch(op) {
515     default:
516       goto error;
517   }
518   case Cronhook:
519     send(x->term, Cldis, 0);
520     send(x, Cldis, 0);
521     x->state = idle;
522     break;
523   case Crnone:
524     x->state = ring;
525     //no state change
526   }
527   //one phone has audible ringing, other phone
528   // is ringing
529   switch(op) {
530     default:
531       goto error;
532   }
533   case Cronhook:
534     send(x->term, Cldis, 0);
535     send(x, Cldis, 0);
536     x->state = idle;
537     break;
538   case Crnone:
539     x->state = ring;
540     //no state change
541   }
542   //one phone has audible ringing, other phone
543   // is ringing
544   switch(op) {
545     default:
546       goto error;
547   }
548   case Cronhook:
549     send(x->term, Cldis, 0);
550     send(x, Cldis, 0);
551     x->state = idle;
552     break;
553   case Crnone:
554     x->state = ring;
555     //no state change
556   }
557   //one phone has audible ringing, other phone
558   // is ringing
559   switch(op) {
560     default:
561       goto error;
562   }
563   case Cronhook:
564     send(x->term, Cldis, 0);
565     send(x, Cldis, 0);
566     x->state = idle;
567     break;
568   case Crnone:
569     x->state = ring;
570     //no state change
571   }
572   //one phone has audible ringing, other phone
573   // is ringing
574   switch(op) {
575     default:
576       goto error;
577   }
578   case Cronhook:
579     send(x->term, Cldis, 0);
580     send(x, Cldis, 0);
581     x->state = idle;
582     break;
583   case Crnone:
584     x->state = ring;
585     //no state change
586   }
587   //one phone has audible ringing, other phone
588   // is ringing
589   switch(op) {
590     default:
591       goto error;
592   }
593   case Cronhook:
594     send(x->term, Cldis, 0);
595     send(x, Cldis, 0);
596     x->state = idle;
597     break;
598   case Crnone:
599     x->state = ring;
600     //no state change
601   }
602   //one phone has audible ringing, other phone
603   // is ringing
604   switch(op) {
605     default:
606       goto error;
607   }
608   case Cronhook:
609     send(x->term, Cldis, 0);
610     send(x, Cldis, 0);
611     x->state = idle;
612     break;
613   case Crnone:
614     x->state = ring;
615     //no state change
616   }
617   //one phone has audible ringing, other phone
618   // is ringing
619   switch(op) {
620     default:
621       goto error;
622   }
623   case Cronhook:
624     send(x->term, Cldis, 0);
625     send(x, Cldis, 0);
626     x->state = idle;
627     break;
628   case Crnone:
629     x->state = ring;
630     //no state change
631   }
632   //one phone has audible ringing, other phone
633   // is ringing
634   switch(op) {
635     default:
636       goto error;
637   }
638   case Cronhook:
639     send(x->term, Cldis, 0);
640     send(x, Cldis, 0);
641     x->state = idle;
642     break;
643   case Crnone:
644     x->state = ring;
645     //no state change
646   }
647   //one phone has audible ringing, other phone
648   // is ringing
649   switch(op) {
650     default:
651       goto error;
652   }
653   case Cronhook:
654     send(x->term, Cldis, 0);
655     send(x, Cldis, 0);
656     x->state = idle;
657     break;
658   case Crnone:
659     x->state = ring;
660     //no state change
661   }
662   //one phone has audible ringing, other phone
663   // is ringing
664   switch(op) {
665     default:
666       goto error;
667   }
668   case Cronhook:
669     send(x->term, Cldis, 0);
670     send(x, Cldis, 0);
671     x->state = idle;
672     break;
673   case Crnone:
674     x->state = ring;
675     //no state change
676   }
677   //one phone has audible ringing, other phone
678   // is ringing
679   switch(op) {
680     default:
681       goto error;
682   }
683   case Cronhook:
684     send(x->term, Cldis, 0);
685     send(x, Cldis, 0);
686     x->state = idle;
687     break;
688   case Crnone:
689     x->state = ring;
690     //no state change
691   }
692   //one phone has audible ringing, other phone
693   // is ringing
694   switch(op) {
695     default:
696       goto error;
697   }
698   case Cronhook:
699     send(x->term, Cldis, 0);
700     send(x, Cldis, 0);
701     x->state = idle;
702     break;
703   case Crnone:
704     x->state = ring;
705     //no state change
706   }
707   //one phone has audible ringing, other phone
708   // is ringing
709   switch(op) {
710     default:
711       goto error;
712   }
713   case Cronhook:
714     send(x->term, Cldis, 0);
715     send(x, Cldis, 0);
716     x->state = idle;
717     break;
718   case Crnone:
719     x->state = ring;
720     //no state change
721   }
722   //one phone has audible ringing, other phone
723   // is ringing
724   switch(op) {
725     default:
726       goto error;
727   }
728   case Cronhook:
729     send(x->term, Cldis, 0);
730     send(x, Cldis, 0);
731     x->state = idle;
732     break;
733   case Crnone:
734     x->state = ring;
735     //no state change
736   }
737   //one phone has audible ringing, other phone
738   // is ringing
739   switch(op) {
740     default:
741       goto error;
742   }
743   case Cronhook:
744     send(x->term, Cldis, 0);
745     send(x, Cldis, 0);
746     x->state = idle;
747     break;
748   case Crnone:
749     x->state = ring;
750     //no state change
751   }
752   //one phone has audible ringing, other phone
753   // is ringing
754   switch(op) {
755     default:
756       goto error;
757   }
758   case Cronhook:
759     send(x->term, Cldis, 0);
760     send(x, Cldis, 0);
761     x->state = idle;
762     break;
763   case Crnone:
764     x->state = ring;
765     //no state change
766   }
767   //one phone has audible ringing, other phone
768   // is ringing
769   switch(op) {
770     default:
771       goto error;
772   }
773   case Cronhook:
774     send(x->term, Cldis, 0);
775     send(x, Cldis, 0);
776     x->state = idle;
777     break;
778   case Crnone:
779     x->state = ring;
780     //no state change
781   }
782   //one phone has audible ringing, other phone
783   // is ringing
784   switch(op) {
785     default:
786       goto error;
787   }
788   case Cronhook:
789     send(x->term, Cldis, 0);
790     send(x, Cldis, 0);
791     x->state = idle;
792     break;
793   case Crnone:
794     x->state = ring;
795     //no state change
796   }
797   //one phone has audible ringing, other phone
798   // is ringing
799   switch(op) {
800     default:
801       goto error;
802   }
803   case Cronhook:
804     send(x->term, Cldis, 0);
805     send(x, Cldis, 0);
806     x->state = idle;
807     break;
808   case Crnone:
809     x->state = ring;
810     //no state change
811   }
812   //one phone has audible ringing, other phone
813   // is ringing
814   switch(op) {
815     default:
816       goto error;
817   }
818   case Cronhook:
819     send(x->term, Cldis, 0);
820     send(x, Cldis, 0);
821     x->state = idle;
822     break;
823   case Crnone:
824     x->state = ring;
825     //no state change
826   }
827   //one phone has audible ringing, other phone
828   // is ringing
829   switch(op) {
830     default:
831       goto error;
832   }
833   case Cronhook:
834     send(x->term, Cldis, 0);
835     send(x, Cldis, 0);
836     x->state = idle;
837     break;
838   case Crnone:
839     x->state = ring;
840     //no state change
841   }
842   //one phone has audible ringing, other phone
843   // is ringing
844   switch(op) {
845     default:
846       goto error;
847   }
848   case Cronhook:
849     send(x->term, Cldis, 0);
850     send(x, Cldis, 0);
851     x->state = idle;
852     break;
853   case Crnone:
854     x->state = ring;
855     //no state change
856   }
857   //one phone has audible ringing, other phone
858   // is ringing
859   switch(op) {
860     default:
861       goto error;
862   }
863   case Cronhook:
864     send(x->term, Cldis, 0);
865     send(x, Cldis, 0);
866     x->state = idle;
867     break;
868   case Crnone:
869     x->state = ring;
870     //no state change
871   }
872   //one phone has audible ringing, other phone
873   // is ringing
874   switch(op) {
875     default:
876       goto error;
877   }
878   case Cronhook:
879     send(x->term, Cldis, 0);
880     send(x, Cldis, 0);
881     x->state = idle;
882     break;
883   case Crnone:
884     x->state = ring;
885     //no state change
886   }
887   //one phone has audible ringing, other phone
888   // is ringing
889   switch(op) {
890     default:
891       goto error;
892   }
893   case Cronhook:
894     send(x->term, Cldis, 0);
895     send(x, Cldis, 0);
896     x->state = idle;
897     break;
898   case Crnone:
899     x->state = ring;
900     //no state change
901   }
902   //one phone has audible ringing, other phone
903   // is ringing
904   switch(op) {
905     default:
906       goto error;
907   }
908   case Cronhook:
909     send(x->term, Cldis, 0);
910     send(x, Cldis, 0);
911     x->state = idle;
912     break;
913   case Crnone:
914     x->state = ring;
915     //no state change
916   }
917   //one phone has audible ringing, other phone
918   // is ringing
919   switch(op) {
920     default:
921       goto error;
922   }
923   case Cronhook:
924     send(x->term, Cldis, 0);
925     send(x, Cldis, 0);
926     x->state = idle;
927     break;
928   case Crnone:
929     x->state = ring;
930     //no state change
931   }
932   //one phone has audible ringing, other phone
933   // is ringing
934   switch(op) {
935     default:
936       goto error;
937   }
938   case Cronhook:
939     send(x->term, Cldis, 0);
940     send(x, Cldis, 0);
941     x->state = idle;
942     break;
943   case Crnone:
944     x->state = ring;
945     //no state change
946   }
947   //one phone has audible ringing, other phone
948   // is ringing
949   switch(op) {
950     default:
951       goto error;
952   }
953   case Cronhook:
954     send(x->term, Cldis, 0);
955     send(x, Cldis, 0);
956     x->state = idle;
957     break;
958   case Crnone:
959     x->state = ring;
960     //no state change
961   }
962   //one phone has audible ringing, other phone
963   // is ringing
964   switch(op) {
965     default:
966       goto error;
967   }
968   case Cronhook:
969     send(x->term, Cldis, 0);
970     send(x, Cldis, 0);
971     x->state = idle;
972     break;
973   case Crnone:
974     x->state = ring;
975     //no state change
976   }
977   //one phone has audible ringing, other phone
978   // is ringing
979   switch(op) {
980     default:
981       goto error;
982   }
983   case Cronhook:
984     send(x->term, Cldis, 0);
985     send(x, Cldis, 0);
986     x->state = idle;
987     break;
988   case Crnone:
989     x->state = ring;
990     //no state change
991   }
992   //one phone has audible ringing, other phone
993   // is ringing
994   switch(op) {
995     default:
996       goto error;
997   }
998   case Cronhook:
999     send(x->term, Cldis, 0);
1000     send(x, Cldis, 0);
1001     x->state = idle;
1002     break;
1003   case Crnone:
1004     x->state = ring;
1005     //no state change
1006   }
1007   //one phone has audible ringing, other phone
1008   // is ringing
1009   switch(op) {
1010     default:
1011       goto error;
1012   }
1013   case Cronhook:
1014     send(x->term, Cldis, 0);
1015     send(x, Cldis, 0);
1016     x->state = idle;
1017     break;
1018   case Crnone:
1019     x->state = ring;
1020     //no state change
1021   }
1022   //one phone has audible ringing, other phone
1023   // is ringing
1024   switch(op) {
1025     default:
1026       goto error;
1027   }
1028   case Cronhook:
1029     send(x->term, Cldis, 0);
1030     send(x, Cldis, 0);
1031     x->state = idle;
1032     break;
1033   case Crnone:
1034     x->state = ring;
1035     //no state change
1036   }
1037   //one phone has audible ringing, other phone
1038   // is ringing
1039   switch(op) {
1040     default:
1041       goto error;
1042   }
1043   case Cronhook:
1044     send(x->term, Cldis, 0);
1045     send(x, Cldis, 0);
1046     x->state = idle;
1047     break;
1048   case Crnone:
1049     x->state = ring;
1050     //no state change
1051   }
1052   //one phone has audible ringing, other phone
1053   // is ringing
1054   switch(op) {
1055     default:
1056       goto error;
1057   }
1058   case Cronhook:
1059     send(x->term, Cldis, 0);
1060     send(x, Cldis, 0);
1061     x->state = idle;
1062     break;
1063   case Crnone:
1064     x->state = ring;
1065     //no state change
1066   }
1067   //one phone has audible ringing, other phone
1068   // is ringing
1069   switch(op) {
1070     default:
1071       goto error;
1072   }
1073   case Cronhook:
1074     send(x->term, Cldis, 0);
1075     send(x, Cldis, 0);
1076     x->state = idle;
1077     break;
1078   case Crnone:
1079     x->state = ring;
1080     //no state change
1081   }
1082   //one phone has audible ringing, other phone
1083   // is ringing
1084   switch(op) {
1085     default:
1086       goto error;
1087   }
1088   case Cronhook:
1089     send(x->term, Cldis, 0);
1090     send(x, Cldis, 0);
1091     x->state = idle;
1092     break;
1093   case Crnone:
1094     x->state = ring;
1095     //no state change
1096   }
1097   //one phone has audible ringing, other phone
1098   // is ringing
1099   switch(op) {
1100     default:
1101       goto error;
1102   }
1103   case Cronhook:
1104     send(x->term, Cldis, 0);
1105     send(x, Cldis, 0);
1106     x->state = idle;
1107     break;
1108   case Crnone:
1109     x->state = ring;
1110     //no state change
1111   }
1112   //one phone has audible ringing, other phone
1113   // is ringing
1114   switch(op) {
1115     default:
1116       goto error;
1117   }
1118   case Cronhook:
1119     send(x->term, Cldis, 0);
1120     send(x, Cldis, 0);
1121     x->state = idle;
1122     break;
1123   case Crnone:
1124     x->state = ring;
1125     //no state change
1126   }
1127   //one phone has audible ringing, other phone
1128   // is ringing
1129   switch(op) {
1130     default:
1131       goto error;
1132   }
1133   case Cronhook:
1134     send(x->term, Cldis, 0);
1135     send(x, Cldis, 0);
1136     x->state = idle;
1137     break;
1138   case Crnone:
1139     x->state = ring;
1140     //no state change
1141   }
1142   //one phone has audible ringing, other phone
1143   // is ringing
1144   switch(op) {
1145     default:
1146       goto error;
1147   }
1148   case Cronhook:
1149     send(x->term, Cldis, 0);
1150     send(x, Cldis, 0);
1151     x->state = idle;
1152     break;
1153   case Crnone:
1154     x->state = ring;
1155     //no state change
1156   }
1157   //one phone has audible ringing, other phone
1158   // is ringing
1159   switch(op) {
1160     default:
1161       goto error;
1162   }
1163   case Cronhook:
1164     send(x->term, Cldis, 0);
1165     send(x, Cldis, 0);
1166     x->state = idle;
1167     break;
1168   case Crnone:
1169     x->state = ring;
1170     //no state change
1171   }
1172   //one phone has audible ringing, other phone
1173   // is ringing
1174   switch(op) {
1175     default:
1176       goto error;
1177   }
1178   case Cronhook:
1179     send(x->term, Cldis, 0);
1180     send(x, Cldis, 0);
1181     x->state = idle;
1182     break;
1183   case Crnone:
1184     x->state = ring;
1185     //no state change
1186   }
1187   //one phone has audible ringing, other phone
1188   // is ringing
1189   switch(op) {
1190     default:
1191       goto error;
1192   }
1193   case Cronhook:
1194     send(x->term, Cldis, 0);
1195     send(x, Cldis, 0);
1196     x->state = idle;
1197     break;
1198   case Crnone:
1199     x->state = ring;
1200     //no state change
1201   }
1202   //one phone has audible ringing, other phone
1203   // is ringing
1204   switch(op) {
1205     default:
1206       goto error;
1207   }
1208   case Cronhook:
1209     send(x->term, Cldis, 0);
1210     send(x, Cldis, 0);
1211     x->state = idle;
1212     break;
1213   case Crnone:
1214     x->state = ring;
1215     //no state change
1216   }
1217   //one phone has audible ringing, other phone
1218   // is ringing
1219   switch(op) {
1220     default:
1221       goto error;
1222   }
1223   case Cronhook:
1224     send(x->term, Cldis, 0);
1225     send(x, Cldis, 0);
1226     x->state = idle;
1227     break;
1228   case Crnone:
1229     x->state = ring;
1230     //no state change
1231   }
1232   //one phone has audible ringing, other phone
1233   // is ringing
1234   switch(op) {
1235     default:
1236       goto error;
1237   }
1238   case Cronhook:
1239     send(x->term, Cldis, 0);
1240     send(x, Cldis, 0);
1241     x->state = idle;
1242     break;
1243   case Crnone:
1244     x->state = ring;
1245     //no state change
1246   }
1247   //one phone has audible ringing, other phone
1248   // is ringing
1249   switch(op) {
1250     default:
1251       goto error;
1252   }
1253   case Cronhook:
1254     send(x->term, Cldis, 0);
1255     send(x, Cldis, 0);
1256     x->state = idle;
1257     break;
1258   case Crnone:
1259     x->state = ring;
1260     //no state change
1261   }
1262   //one phone has audible ringing, other phone
1263   // is ringing
1264   switch(op) {
1265     default:
1266       goto error;
1267   }
1268   case Cronhook:
1269     send(x->term, Cldis, 0);
1270     send(x, Cldis, 0);
1271     x->state = idle;
1272     break;
1273   case Crnone:
1274     x->state = ring;
1275     //no state change
1276   }
1277   //one phone has audible ringing, other phone
1278   // is ringing
1279   switch(op) {
1280     default:
1281       goto error;
1282   }
1283   case Cronhook:
1284     send(x->term, Cldis, 0);
1285     send(x, Cldis, 0);
1286     x->state = idle;
1287     break;
1288   case Crnone:
1289     x->state = ring;
1290     //no state change
1291   }
12
```

FIG. 4





```

191 FROM FIG 3
192
193 Transwer: //cross connect the phones
194 send(x, Ciconn, x->term->lineno);
195 send(x->term, Ciconn, x->lineno);
196 x->state = conn;
197 break;
198
199 break;
200
201 ...
202
203 #include <u.h>
204
205 ...
206
207 69 */
208 70 @dead
209 71 switch(op) {
210 72 default:
211 73 print("%L dead and op=%s\n", x, istring(op));
212 74 goto Bdead;
213 75
214 case Cprovinc: //line is successfully provisioned
215 76 break;
216 77 }
217 x->term = 0;
218 80 /*
219 81 /s
220
221 ...
222
223 84 @idle:
224 85 switch(op) {
225 86 default:
226 87 print("%L idle and op=%s\n", x, istring(op));
227 88 goto Bidle;
228 89
229 case Crdtmf: //noise (dtmf has been removed)
230 90 case Cdis: //noise (line has been disconnected)
231 case Cronhook: //noise (line has been hung up)
232 case Crtone: //noise (tone has been applied)
233 94 goto Bidle;
234
235 ...
236
237 95 case Cronhook:
238 96 break;
239 97
240 98 }
241 99 x->ndigits = 0;
242 100 send(x, Cldtmf, 1);
243
244 ...
245
246 102 @: switch(op) {
247 103 default:
248 104 goto error;
249 105
250 106 case Cronhook: //connect digit detector
251 107 send(x, Cdis, 0);
252 108 goto Bidle;
253 109
254 110 case Crdtmf: //digit detector is active
255
256 ...
257
258 112 @: switch(op) {
259 113 default:
260 114 goto error; //line was hung up
261 115 case Cronhook:
262 116 send(x, Cdis, 0);
263 117 goto Bidle;
264 118 case Crdtmf: //first digit (even before giving dial tone)
265 119 goto Adigit; //dial tone has been applied
266 120 case Crtone:
267 121 break;
268 122 }
269 123
270 ...
271
272 125 @digits
273 126 x->time = seconds(6);
274 127 switch(op) {
275 128 default:
276 129 goto error;
277 130
278 131 case Cronhook: //line was hung up
279 132 send(x, Cdis, 0);
280 133 goto Bidle;
281 134
282 ...
283
284 510 {
285 511
286 ...
287
288 515 {
289 516
290 ...
291
292 530 {
293 531
294 ...
295
296 535 {
297 536
298 ...
299
300 540 {
301 541
302 ...
303
304 545 {
305 546
306 ...
307
308 550 {
309 551
310 ...
311
312 555 {
313 556
314 ...
315
316 560 {
317 561
318 ...
319
320 565 {
321 566
322 ...
323
324 570 {
325 571
326 ...
327
328 575 {
329 576
330 ...
331
332 580 {
333 581
334 ...
335
336 585 {
337 586
338 ...
339
340 590 {
341 591
342 ...
343
344 595 {
345 596
346 ...
347
348 600 {
349 601
350 ...
351
352 605 {
353 606
354 ...
355
356 610 {
357 611
358 ...
359
360 615 {
361 616
362 ...
363
364 620 {
365 621
366 ...
367
368 625 {
369 626
370 ...
371
372 630 {
373 631
374 ...
375
376 635 {
377 636
378 ...
379
380 640 {
381 641
382 ...
383
384 645 {
385 646
386 ...
387
388 650 {
389 651
390 ...
391
392 655 {
393 656
394 ...
395
396 660 {
397 661
398 ...
399
400 665 {
401 666
402 ...
403
404 670 {
405 671
406 ...
407
408 675 {
409 676
410 ...
411
412 680 {
413 681
414 ...
415
416 685 {
417 686
418 ...
419
420 690 {
421 691
422 ...
423
424 695 {
425 696
426 ...
427
428 700 {
429 701
430 ...
431
432 705 {
433 706
434 ...
435
436 710 {
437 711
438 ...
439
440 715 {
441 716
442 ...
443
444 720 {
445 721
446 ...
447
448 725 {
449 726
450 ...
451
452 730 {
453 731
454 ...
455
456 735 {
457 736
458 ...
459
460 740 {
461 741
462 ...
463
464 745 {
465 746
466 ...
467
468 750 {
469 751
470 ...
471
472 755 {
473 756
474 ...
475
476 760 {
477 761
478 ...
479
480 765 {
481 766
482 ...
483
484 770 {
485 771
486 ...
487
488 775 {
489 776
490 ...
491
492 780 {
493 781
494 ...
495
496 785 {
497 786
498 ...
499
500 790 {
501 791
502 ...
503
504 795 {
505 796
506 ...
507
508 800 {
509 801
510 ...
511
512 805 {
513 806
514 ...
515
516 810 {
517 811
518 ...
519
520 815 {
521 816
522 ...
523
524 820 {
525 826
526 ...
527
528 825 {
529 830
530 ...
531
532 835 {
533 836
534 ...
535
536 840 {
537 841
538 ...
539
540 845 {
541 846
542 ...
543
544 850 {
545 851
546 ...
547
548 855 {
549 856
550 ...
551
552 860 {
553 861
554 ...
555
556 865 {
557 866
558 ...
559
560 870 {
561 871
562 ...
563
564 875 {
565 876
566 ...
567
568 880 {
569 881
570 ...
571
572 885 {
573 886
574 ...
575
576 890 {
577 891
578 ...
579
580 895 {
581 896
582 ...
583
584 900 {
585 896
586 ...
587
588 905 {
589 896
590 ...
591
592 910 {
593 896
594 ...
595
596 915 {
597 896
598 ...
599
600 920 {
601 896
602 ...
603
604 925 {
605 896
606 ...
607
608 930 {
609 896
610 ...
611
612 935 {
613 896
614 ...
615
616 940 {
617 896
618 ...
619
620 945 {
621 896
622 ...
623
624 950 {
625 896
626 ...
627
628 955 {
629 896
630 ...
631
632 960 {
633 896
634 ...
635
636 965 {
637 896
638 ...
639
640 970 {
641 896
642 ...
643
644 975 {
645 896
646 ...
647
648 980 {
649 896
650 ...
651
652 985 {
653 896
654 ...
655
656 990 {
657 896
658 ...
659
660 995 {
661 896
662 ...
663
664 1000 {
665 896
666 ...
667
668 1005 {
669 896
670 ...
671
672 1010 {
673 896
674 ...
675
676 1015 {
677 896
678 ...
679
680 1020 {
681 896
682 ...
683
684 1025 {
685 896
686 ...
687
688 1030 {
689 896
690 ...
691
692 1035 {
693 896
694 ...
695
696 1040 {
697 896
698 ...
699
699 1045 {
700 896
701 ...
702
700 1050 {
701 896
702 ...
703
701 1055 {
702 896
703 ...
704
702 1060 {
703 896
704 ...
705
703 1065 {
704 896
705 ...
706
704 1070 {
705 896
706 ...
707
705 1075 {
706 896
707 ...
708
706 1080 {
707 896
708 ...
709
707 1085 {
708 896
709 ...
710
708 1090 {
709 896
710 ...
711
709 1095 {
710 896
711 ...
712
710 1100 {
711 896
712 ...
713
711 1105 {
712 896
713 ...
714
712 1110 {
713 896
714 ...
715
713 1
```

FIG. 6

```

135 FROM FIG 5
136 case Crtone:
137     goto Bidle;
138 case Cralg:
139     /*decode the digits according to the dial plan*/
140     p = dialplan(x->digits, x->ndigits);
141     if(x->ndigits == 1)
142         send(x, Crtone, 0);
143     if(p == 0)
144         goto Bdigits;
145     break;
146 case Crttime:
147     send(x, Cldis, 0);
148     send(x, Crtone, 1scream);
149     goto Bbusy;
150
151 /* translate a local number to a local phone */
152 x->term == 0;
153 if(x->term == 0)
154     send(x, Crtone, 1reorder);
155     goto Bbusy;
156
157 send(x->term, Crtone, 1);
158 //ring the destination
159
160 switch(op) {
161     default:
162         goto error;
163
164 case Cronhook:
165     send(x->term, Cldis, 0);
166     send(x, Cldis, 0);
167     goto Bidle;
168
169 case Crtbusy:
170     send(x, Crtone, Tbusy);
171     goto Bbusy;
172
173 case Crring:
174     //is ringing
175     break;
176     send(x, Crtone, Tring);
177     //apply audible ringing tone
178 }
179
180 @ring:
181 switch(op) {
182     default:
183         goto error;
184
185 case Cronhook:
186     send(x->term, Cldis, 0);
187     send(x, Cldis, 0);
188     goto Bidle;
189
190 case Crtone:
191     goto Bring;
192
193 case Crtanswer:
194     break;
195
196 send(x, Crtconn, x->term->lineno);
197 send(x->term, Crtconn, x->lineno);
198 //cross connect the phones
199
200 @conn:
201 switch(op) {
202     default:
203         case Crtconn;
204         goto Bconn;
205
206 case Cronhook:
207     send(x->term, Cldis, 0);
208     send(x, Cldis, 0);
209     goto Bidle;
210
211 goto error;
212 }
213
214 TO FIG 7

```



FIG. 8

```

FROM FIG 7
141 send(x, Cldis, 0)
142 goto Bidle;
...
192 /*one phone has audible ringing
193 */
194 goto Bring; Bring: x->state = 7; goto out; Aring;;
195 #line 181 sample.z
196
197 switch(op) {
198 default:
199 goto error;
200
201 case Cronhook:
202 send(x->term, Cldis, 0);
203 send(x, Cldis, 0);
204 goto Bidle;
205
206 case Crdone:
207 goto Bring;
208
209 case Cranswer:
210 break;
211 }
212
213 send(x, Cconn, x->term->lineno); //cross connect the phones
214 send(x->term, Cconn, x->lineno);
215 /*
216 *phones are connected
217 */
218
219 goto Bconn; x->state = 8; goto out; Aconn;;
220 #line 204 sample.z
221
222
223
292
293 }

```

© 1998 Lucent Technologies Inc.

FIG. 9

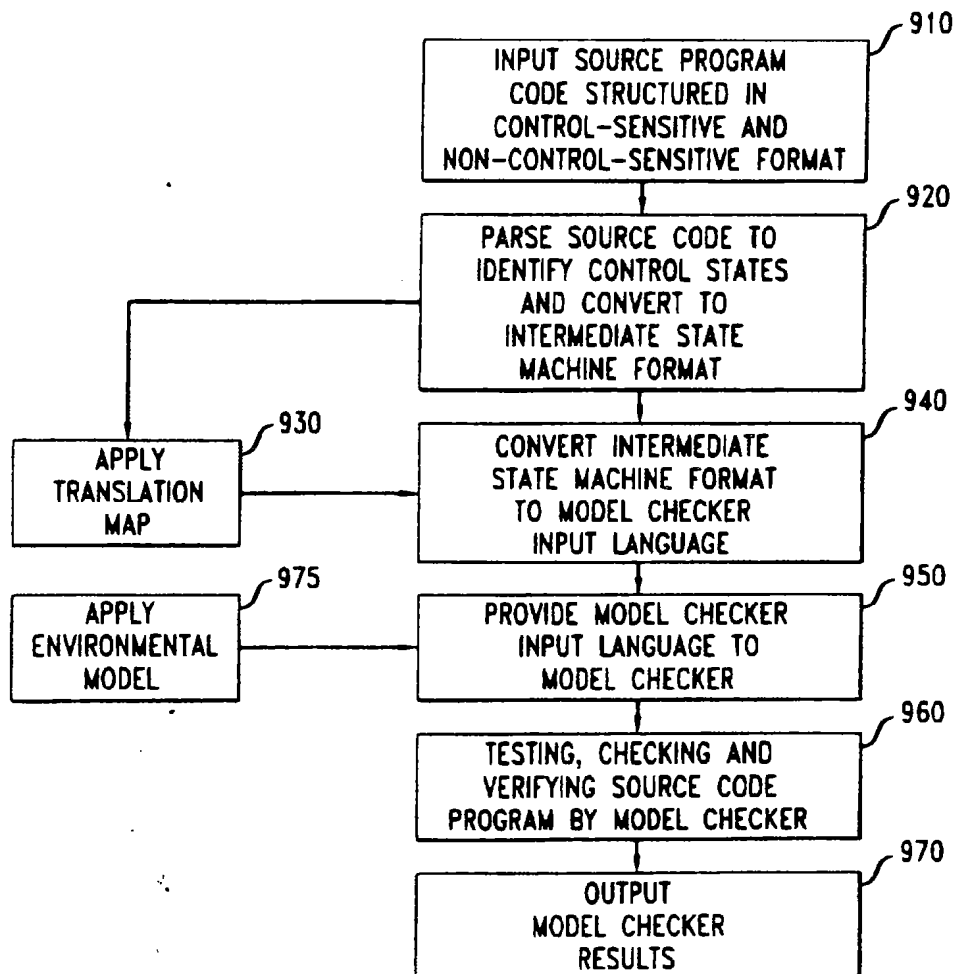


FIG. 10

```

5 %p_template "state_p.tpl"
6 %p_map state.map
7
8 %event Cranswer
9 %event Crbusy
10 %event Crconn
11 %event Crdigit
12 %event Crdis
13 %event Crdml
14 %event Croffhook
15 %event Cronhook
16 %event Crprovc
17 %event Cring
18 %event Crtime
19 %event Crtone
20
21 %state S_dead
22 %state S_idle
23 %state S_idle_1
24 %state S_idle_2
25 %state S_digits
26 %state S_digits_1
27 %state S_ring
28 %state S_conn
29 %state S_busy
30
31
32 S_dead
33 {
34 | Crprovc S_idle
35 {
36 {
37 x->term = 0; /* #79 */; }
38 }
39 }
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63 S_digits: onreceipt
64 {
65 x->time=seconds(6); /* #126 */; }
66 }
67 else error
68 {
69 Cronhook S_idle
70 {
71 send(x,Cdis,0); /* #132 */; }
72 }
73 }
74 }
75 }
76 }
77 }
78 }
79 }
80 }
81 }
82 }
83 }
84 }
85 }
86 }
87 }
88 }
89 }
90 }
91 }
92 }
93 }
94 }
95 }
96 }
97 }
98 }
99 }
100 }
101 }
102 }
103 }
104 }
105 }
106 }
107 }
108 }
109 }
110 }
111 }
112 }
113 }
114 }
115 }
116 }
117 }
118 }
119 }
120 }
121 }
122 }
123 }
124 }
125 }
126 }
127 }
128 }
129 }
130 }
131 }
132 }
133 }
134 }
135 }
136 }
137 }
138 }
139 }
140 }
141 }
142 }
143 }
144 }
145 }
146 }
147 }
148 }
149 }
150 }
151 }
152 }
153 }
154 }
155 }
156 }
157 }
158 }
159 }
160 }
161 }
162 }
163 }
164 }
165 }
166 }
167 }
168 }
169 }
170 }
171 }
172 }
173 }
174 }
175 }
176 }
177 }
178 }
179 }
180 }
181 }
182 }
183 }
184 }
185 }
186 }
187 }
188 }
189 }
190 }
191 }
192 }
193 }
194 }
195 }
196 }
197 }
198 }
199 }
200 }
201 }
202 }
203 }
204 }
205 }
206 }
207 }
208 }
209 }
210 }
211 }
212 }
213 }
214 }
215 }
216 }
217 }
218 }
219 }
220 }
221 }
222 }
223 }
224 }
225 }
226 }
227 }
228 }
229 }
230 }
231 }
232 }
233 }
234 }
235 }
236 }
237 }
238 }
239 }
240 }
241 }
242 }
243 }
244 }
245 }
246 }
247 }
248 }
249 }
250 }
251 }
252 }
253 }
254 }
255 }
256 }
257 }
258 }
259 }
260 }
261 }
262 }
263 }
264 }
265 }
266 }
267 }
268 }
269 }
270 }
271 }
272 }
273 }
274 }
275 }
276 }
277 }
278 }
279 }
280 }
281 }
282 }
283 }
284 }
285 }
286 }
287 }
288 }
289 }
290 }
291 }
292 }
293 }
294 }
295 }
296 }
297 }
298 }
299 }
300 }
301 }
302 }
303 }
304 }
305 }
306 }
307 }
308 }
309 }
310 }
311 }
312 }
313 }
314 }
315 }
316 }
317 }
318 }
319 }
320 }
321 }
322 }
323 }
324 }
325 }
326 }
327 }
328 }
329 }
330 }
331 }
332 }
333 }
334 }
335 }
336 }
337 }
338 }
339 }
340 }
341 }
342 }
343 }
344 }
345 }
346 }
347 }
348 }
349 }
350 }
351 }
352 }
353 }
354 }
355 }
356 }
357 }
358 }
359 }
360 }
361 }
362 }
363 }
364 }
365 }
366 }
367 }
368 }
369 }
370 }
371 }
372 }
373 }
374 }
375 }
376 }
377 }
378 }
379 }
380 }
381 }
382 }
383 }
384 }
385 }
386 }
387 }
388 }
389 }
390 }
391 }
392 }
393 }
394 }
395 }
396 }
397 }
398 }
399 }
400 }
401 }
402 }
403 }
404 }
405 }
406 }
407 }
408 }
409 }
410 }
411 }
412 }
413 }
414 }
415 }
416 }
417 }
418 }
419 }
420 }
421 }
422 }
423 }
424 }
425 }
426 }
427 }
428 }
429 }
430 }
431 }
432 }
433 }
434 }
435 }
436 }
437 }
438 }
439 }
440 }
441 }
442 }
443 }
444 }
445 }
446 }
447 }
448 }
449 }
450 }
451 }
452 }
453 }
454 }
455 }
456 }
457 }
458 }
459 }
460 }
461 }
462 }
463 }
464 }
465 }
466 }
467 }
468 }
469 }
470 }
471 }
472 }
473 }
474 }
475 }
476 }
477 }
478 }
479 }
480 }
481 }
482 }
483 }
484 }
485 }
486 }
487 }
488 }
489 }
490 }
491 }
492 }
493 }
494 }
495 }
496 }
497 }
498 }
499 }
500 }
501 }
502 }
503 }
504 }
505 }
506 }
507 }
508 }
509 }
510 }
511 }
512 }
513 }
514 }
515 }
516 }
517 }
518 }
519 }
520 }
521 }
522 }
523 }
524 }
525 }
526 }
527 }
528 }
529 }
530 }
531 }
532 }
533 }
534 }
535 }
536 }
537 }
538 }
539 }
540 }
541 }
542 }
543 }
544 }
545 }
546 }
547 }
548 }
549 }
550 }
551 }
552 }
553 }
554 }
555 }
556 }
557 }
558 }
559 }
560 }
561 }
562 }
563 }
564 }
565 }
566 }
567 }
568 }
569 }
570 }
571 }
572 }
573 }
574 }
575 }
576 }
577 }
578 }
579 }
580 }
581 }
582 }
583 }
584 }
585 }
586 }
587 }
588 }
589 }
590 }
591 }
592 }
593 }
594 }
595 }
596 }
597 }
598 }
599 }
600 }
601 }
602 }
603 }
604 }
605 }
606 }
607 }
608 }
609 }
610 }
611 }
612 }
613 }
614 }
615 }
616 }
617 }
618 }
619 }
620 }
621 }
622 }
623 }
624 }
625 }
626 }
627 }
628 }
629 }
630 }
631 }
632 }
633 }
634 }
635 }
636 }
637 }
638 }
639 }
640 }
641 }
642 }
643 }
644 }
645 }
646 }
647 }
648 }
649 }
650 }
651 }
652 }
653 }
654 }
655 }
656 }
657 }
658 }
659 }
660 }
661 }
662 }
663 }
664 }
665 }
666 }
667 }
668 }
669 }
670 }
671 }
672 }
673 }
674 }
675 }
676 }
677 }
678 }
679 }
680 }
681 }
682 }
683 }
684 }
685 }
686 }
687 }
688 }
689 }
690 }
691 }
692 }
693 }
694 }
695 }
696 }
697 }
698 }
699 }
700 }
701 }
702 }
703 }
704 }
705 }
706 }
707 }
708 }
709 }
710 }
711 }
712 }
713 }
714 }
715 }
716 }
717 }
718 }
719 }
720 }
721 }
722 }
723 }
724 }
725 }
726 }
727 }
728 }
729 }
730 }
731 }
732 }
733 }
734 }
735 }
736 }
737 }
738 }
739 }
740 }
741 }
742 }
743 }
744 }
745 }
746 }
747 }
748 }
749 }
750 }
751 }
752 }
753 }
754 }
755 }
756 }
757 }
758 }
759 }
760 }
761 }
762 }
763 }
764 }
765 }
766 }
767 }
768 }
769 }
770 }
771 }
772 }
773 }
774 }
775 }
776 }
777 }
778 }
779 }
780 }
781 }
782 }
783 }
784 }
785 }
786 }
787 }
788 }
789 }
790 }
791 }
792 }
793 }
794 }
795 }
796 }
797 }
798 }
799 }
800 }
801 }
802 }
803 }
804 }
805 }
806 }
807 }
808 }
809 }
810 }
811 }
812 }
813 }
814 }
815 }
816 }
817 }
818 }
819 }
820 }
821 }
822 }
823 }
824 }
825 }
826 }
827 }
828 }
829 }
830 }
831 }
832 }
833 }
834 }
835 }
836 }
837 }
838 }
839 }
840 }
841 }
842 }
843 }
844 }
845 }
846 }
847 }
848 }
849 }
850 }
851 }
852 }
853 }
854 }
855 }
856 }
857 }
858 }
859 }
860 }
861 }
862 }
863 }
864 }
865 }
866 }
867 }
868 }
869 }
870 }
871 }
872 }
873 }
874 }
875 }
876 }
877 }
878 }
879 }
880 }
881 }
882 }
883 }
884 }
885 }
886 }
887 }
888 }
889 }
890 }
891 }
892 }
893 }
894 }
895 }
896 }
897 }
898 }
899 }
900 }
901 }
902 }
903 }
904 }
905 }
906 }
907 }
908 }
909 }
910 }
911 }
912 }
913 }
914 }
915 }
916 }
917 }
918 }
919 }
920 }
921 }
922 }
923 }
924 }
925 }
926 }
927 }
928 }
929 }
930 }
931 }
932 }
933 }
934 }
935 }
936 }
937 }
938 }
939 }
940 }
941 }
942 }
943 }
944 }
945 }
946 }
947 }
948 }
949 }
950 }
951 }
952 }
953 }
954 }
955 }
956 }
957 }
958 }
959 }
960 }
961 }
962 }
963 }
964 }
965 }
966 }
967 }
968 }
969 }
970 }
971 }
972 }
973 }
974 }
975 }
976 }
977 }
978 }
979 }
980 }
981 }
982 }
983 }
984 }
985 }
986 }
987 }
988 }
989 }
990 }
991 }
992 }
993 }
994 }
995 }
996 }
997 }
998 }
999 }
1000 }

```

FIG. 11

```

1# Names of events introduced only here:
2
3 %event Ciconn
4 %event Cldis
5 %event Cldlmt
6 %event Clring
7 %event Clanswer
8 %event Cbusy
9
10# Statements that are not the target of this check:
11
12 p=dialplan(x->digits, x->ndigits) true /*not target of this check*/
13 x->ndigits=0 true /*ditto*/
14 x->term=0 true
15 x->term=lookup(p) true
16 print true
17 send(x, Cltone, 0) true
18 send(x, Cltone, Cbusy) true
19 send(x, Cltone, Idial) true
20 send(x, Cltone, Irearpt) true
21 send(x, Cltone, Iring) true
22 send(x, Cltone, Iscream) true
23
24# allow these properties to be assumed either true
25# or false (introducing non-determinism in the abstraction):
26
27 (x->ndigits==1) true /*assume condition true*/
28 !((x->ndigits==1)) true /* or its negation true */
29
30 (p==0) true /*ditto*/
31
32
33
34
35
36# Statements that are relevant and directly represented:
37
38 x->time=seconds(6) Timer = true /*start timeout timer*/
39 send(x, Ciconn, x->term->lineno) acs:Ciconn /*send q message*/
40 send(x, Cldis, b) acs:Cldis
41 send(x, Cldlmt, 1) acs:Cldlmt
42 send(x->term, Ciconn, x->lineno) acs:Ciconn
43 send(x->term, Cldis, 0) acs:Cdis
44
45# Nondeterministically abstract in one of two ways:
46
47 send(x->term, Clring, 1) if :: acs:Clring; acs:Clanswer
:: acs:Cbusy T1

```

*FIG. 12*

1200

```
1  /*  
2  * source: state */  
:  
139 A_S_digits_Cronhook;  
140 /* { send(x, Cldis, 0); ** #132 **; } */  
↓  
TO FIG 13
```

© 1998 Lucent Technologies Inc.



*FIG. 13*

FROM FIG 12

1200

```
↓
141 goto BS_idle;
142 :: else ->
    .
302 qin'Crprovlc      /*provision the phone*/
303 }
```

© 1998 Lucent Technologies Inc.

*FIG. 14*

1400

```
1  /*  
2  *source: state.x  
:  
152 od  
153 }
```

© 1998 Lucent Technologies Inc.

FIG. 15

1500

```

1  /* Spin template */
2  :
3  :
19 prototype context ()
20 { mtype msg;
21 :
22 end: do
23 :: Timer && timeout -> /* the timer expires */
24   Timer = false;      /* turn it off */
25   qin!Crtime          /* notify the switch */
26 :
27 :: atomic {
28   acs?msg->           /* message sent to environment */
29   Timer = false;     /* on arrival of any event: timer cancels */
30 :
31   if
32   :: msg == Ctdtmf -> /* digit recognition */
33     qin!Crdtmf;      /* must acknowledge this event */
34     dialtone = 1 - dialtone /* toggle on-off */
35 :
36   :: msg == Ctdis ->
37     qin!Crdis;       /* acknowledge */
38     dialtone = false /* disconnecting - dialtone off */
39 :
40   :: msg == Ctring -> /* phone rings */
41     qin!Crring
42 :
43   :: msg == Ctanswer -> /* phone answered */
44     qin!Cranswer
45 :
46   :: msg == Ctbusy -> /* phone is busy */
47     qin!Crbusy
48 :
49   :: msg == Crdis    msg == Ctconn    msg == Crconn ->
50     skip /* messages not yet considered */
51 :
52   :: else -> /* nothing else should happen */
53     printf("context: cannot happen\n");
54     assert(false) /* assert that we cannot reach this */
55 :
56   fi
57 } od
58 :
70 od

```

1510 {

FIG. 16

1600

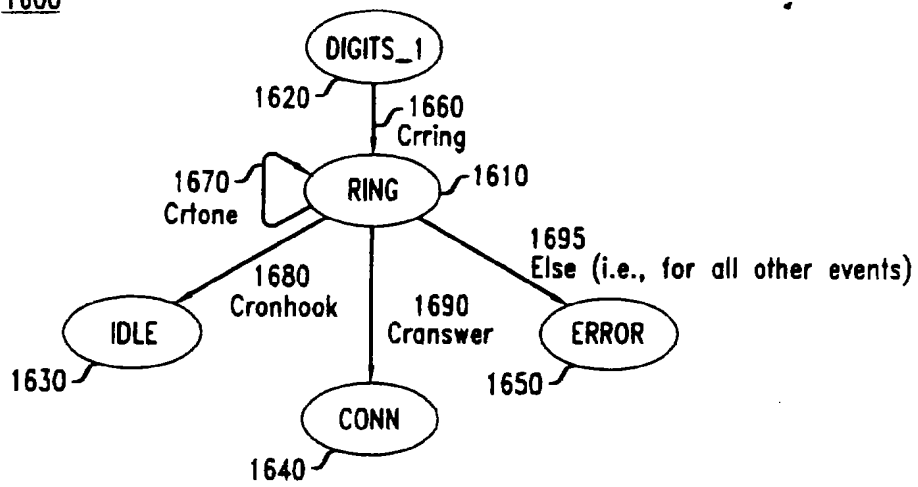


FIG. 17

1700

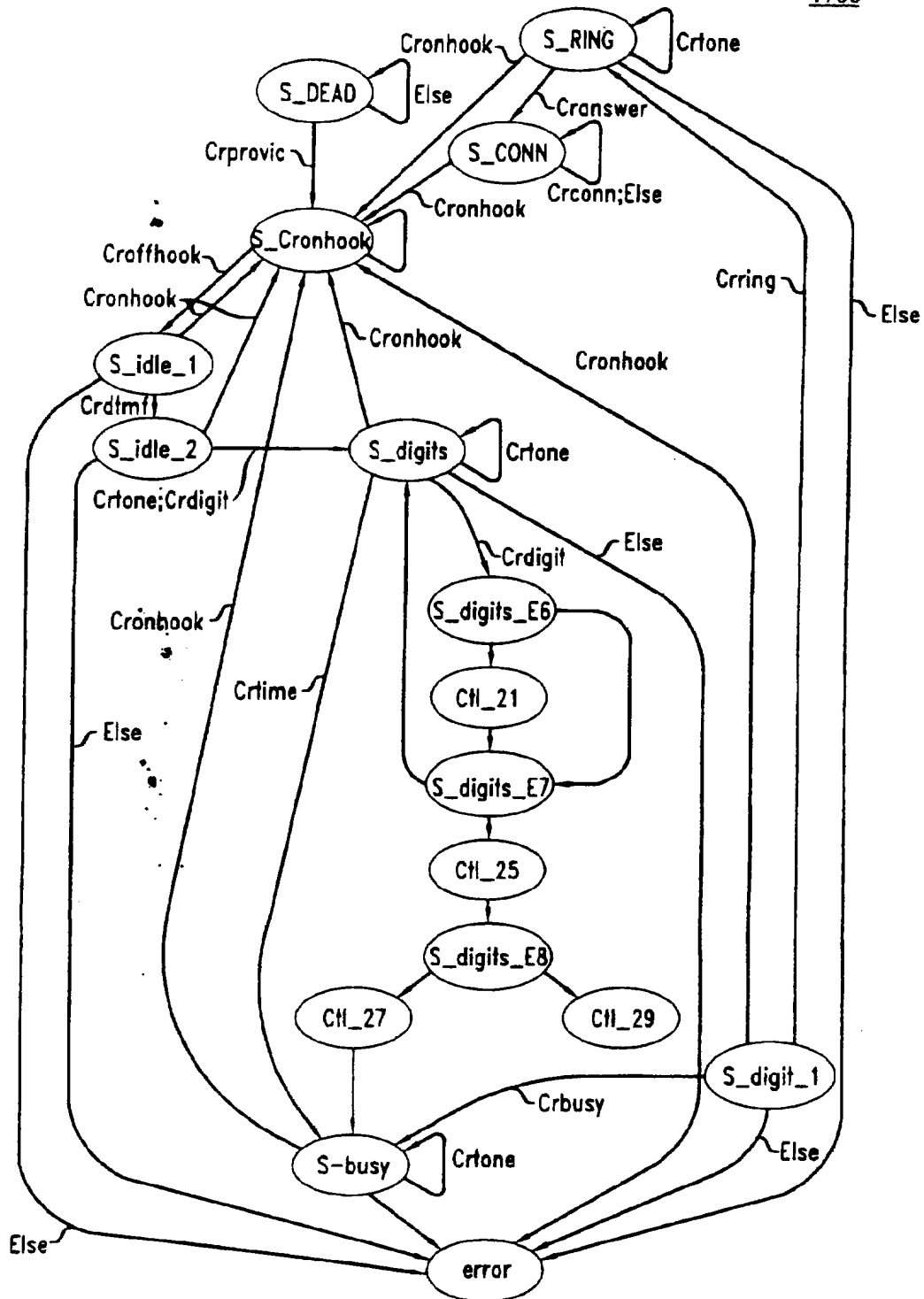


FIG. 18

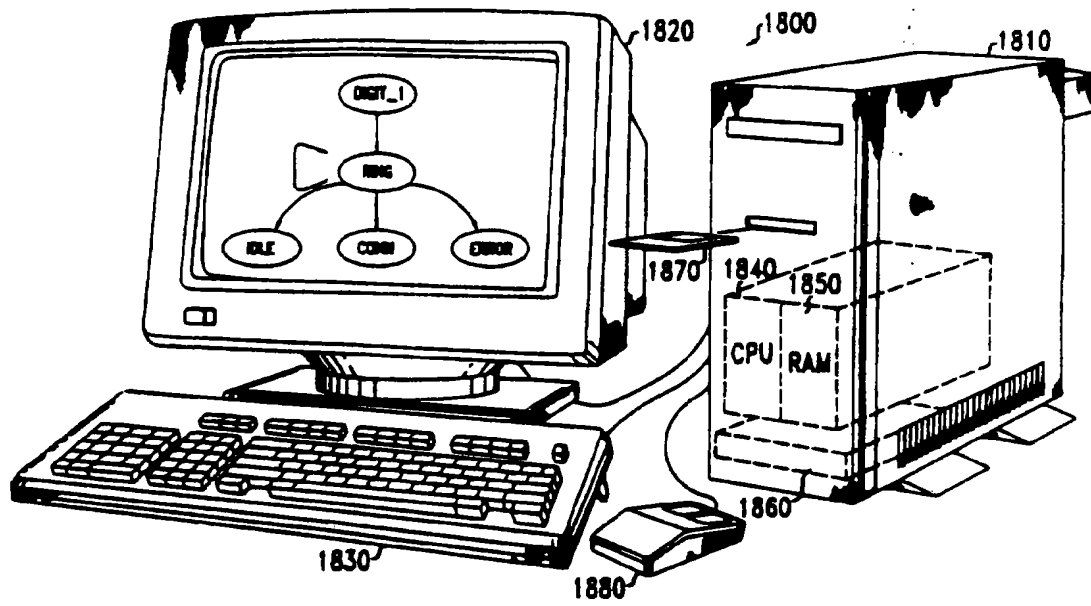


FIG. 19

1900

```

1 Error scenario for specification.
2 Reported by a Spin verification
3
4
5 < state S_dead >          /* control state */
6 Crprovc                  /* event received */
7
8 {x->term=0; ** #79 **; }
9
10 < state S_idle >          /* C code executed */
11 Croffhook                /* etc. */
12
13 { x->ndigits=0; ** #99 **; send(x,Cldtmf,1); ** #100 **; }
14
15 < state S_idle_1 >
16 Crdtmf
17
18 { send(x,Cftone,Tdial); ** #111 **; )
19
20 < state S_idle_2 >
21 Crdigit
22
23 { x->time=seconds(6); ** #126 **; }
1510 { 24 p=dialplan(x->digits,x->ndigits); ** 140 **; }
25 { ((x->ndigits == 1))
26 { ((p == 0))
27 { x->term=lookup(p); **154 **; }
28 { x->term == 0 }
29 { send(x,Cftone,Treorder); ** #156 **; }
30
31 < state S_busy >          /* unspecified event */
32 Crdigit
33
34 reached error state
35
36 spin: line 239 "state.p" Error: assertion violated
37 spin: trail ends after 79 steps
38 #processes: 5
39     queue 3 (acs):
40         dialtone = 1
41         Timer = 1
42 79: proc 4 (context) line 245 "state.p" (state 26) <valid endstate>
43 79: proc 3 (subscriber) line 288 "state.p" (state 9)
44 79: proc 2 (state) line 240 "state.p" (state 189) <valid endstate>
45 79: proc 1 (:int:) line 303 "state.p" (state 6) <valid endstate>
46 79: proc 0 (talker) line 138 "state.p" (state 145) <valid endstate>
47 5 processes created

```

1920



European Patent  
Office

# EUROPEAN SEARCH REPORT

Application Number  
EP 99 30 9818

DOCUMENTS CONSIDERED TO BE RELEVANT			
Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.7)
X	EP 0 685 792 A (AT&T CORP.) 6 December 1995 (1995-12-06) * column 1, line 35 - line 56 *	1-18	G06F11/00
A	GERARD J. HOLZMANN: "The Model Checker SPIN" IEEE TRANSACTIONS ON SOFTWARE ENGINEERING., vol. 23, no. 5, May 1997 (1997-05), pages 279-295, XP000720371 IEEE INC. NEW YORK., US ISSN: 0098-5589 * page 280, left-hand column, line 21 - right-hand column, line 19 *	1-18	
			TECHNICAL FIELDS SEARCHED (Int.Cl.7)
			G06F
The present search report has been drawn up for all claims			
Place of search <b>THE HAGUE</b>		Date of completion of the search <b>3 May 2000</b>	Examiner <b>Corremans, G</b>
CATEGORY OF CITED DOCUMENTS X: particularly relevant if taken alone Y: particularly relevant if combined with another document of the same category A: technological background O: non-written disclosure P: intermediate document		T: theory or principle underlying the invention E: earlier patent document, but published on, or after the filing date D: document cited in the application L: document cited for other reasons &: member of the same patent family, corresponding document	

EPO FORM 1606 08/02 (P04001)



**ANNEX TO THE EUROPEAN SEARCH REPORT  
ON EUROPEAN PATENT APPLICATION NO.**

EP 99 30 9818

This annex lists the patent family members relating to the patent documents cited in the above-mentioned European search report. The members are as contained in the European Patent Office EDP file on  
The European Patent Office is in no way liable for these particulars which are merely given for the purpose of information.

03-05-2000

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
EP 685792 A	06-12-1995	CA 2147536 A	02-12-1995
		JP 7334566 A	22-12-1995
		US 5615137 A	25-03-1997

EPO FORM P4559

For more details about this annex : see Official Journal of the European Patent Office, No. 12/82